Overview

Backpropogation

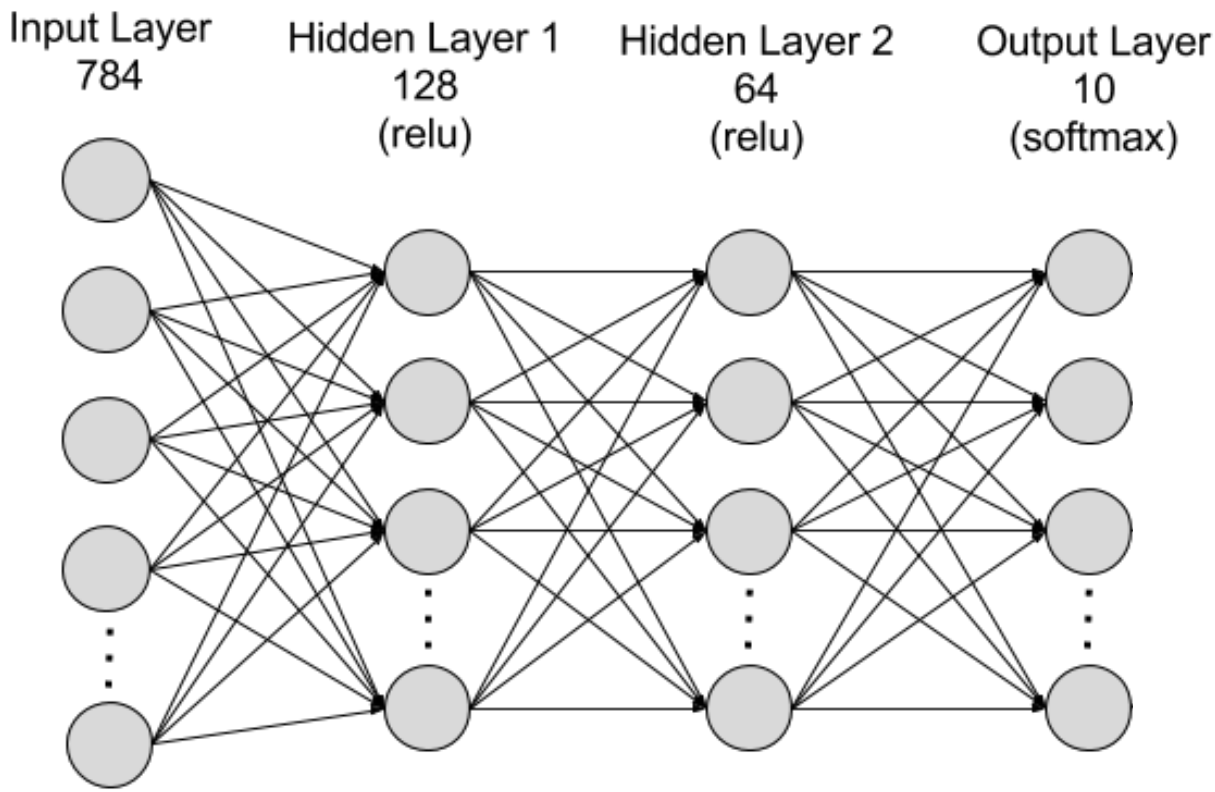
Tips on network training

Training curves

Dropout

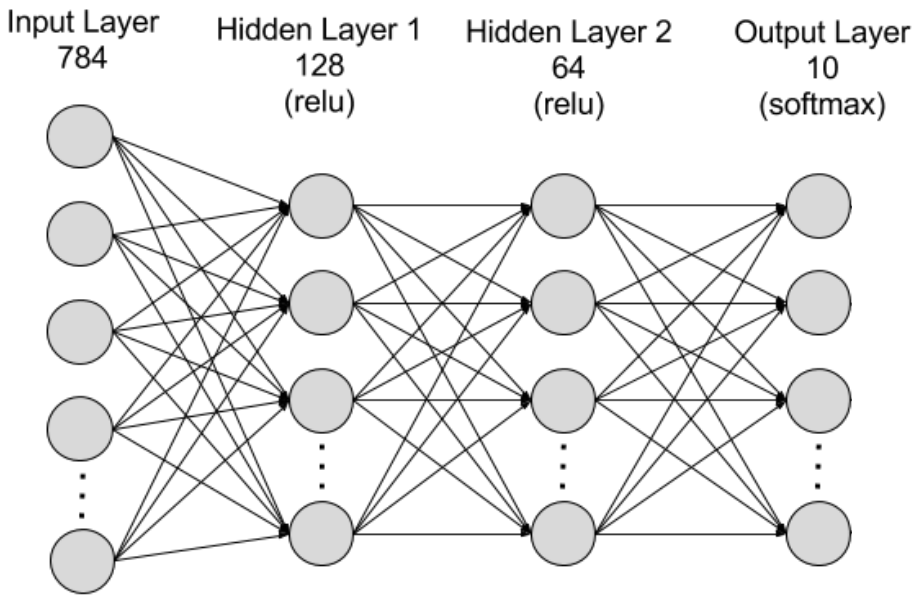
Data augmentation

Hyperparameters


Input Layer
784
Hidden Layer 1
128
(relu)
Hidden Layer 2
64
(relu)
Output Layer
10
(softmax)

Many Parameters

Such networks have **many** parameters!



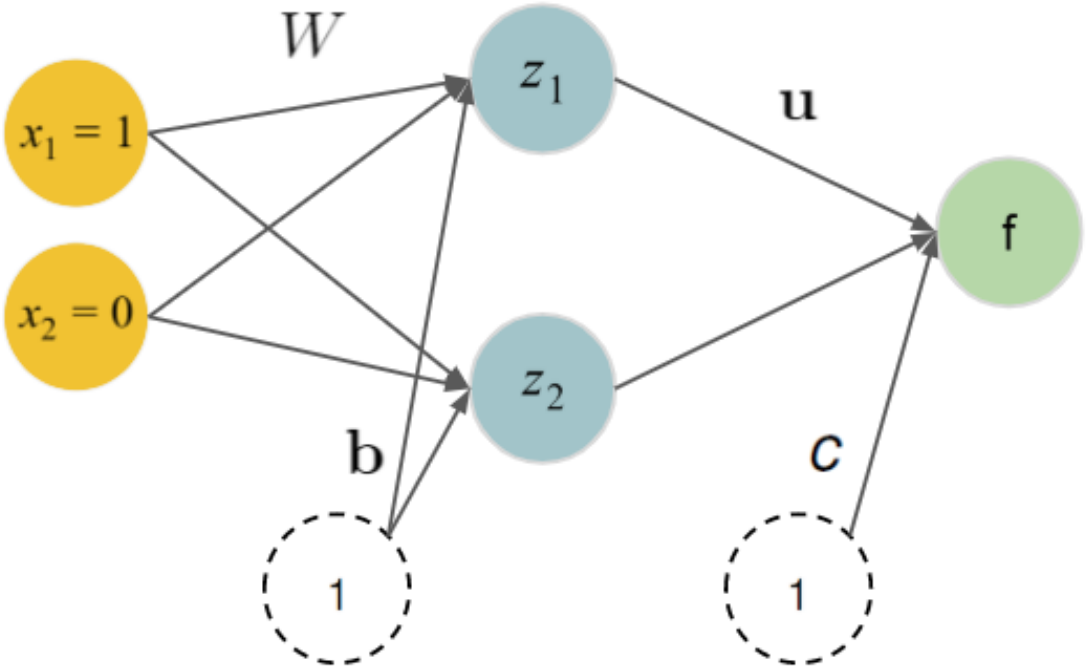
$W^{(1)}$ $b^{(1)}$ $W^{(2)}$ $b^{(2)}$ $W^{(3)}$ $b^{(3)}$

784x128 128 128x64 64 64x10 10

= 109.386 parameters

XOR EXAMPLE

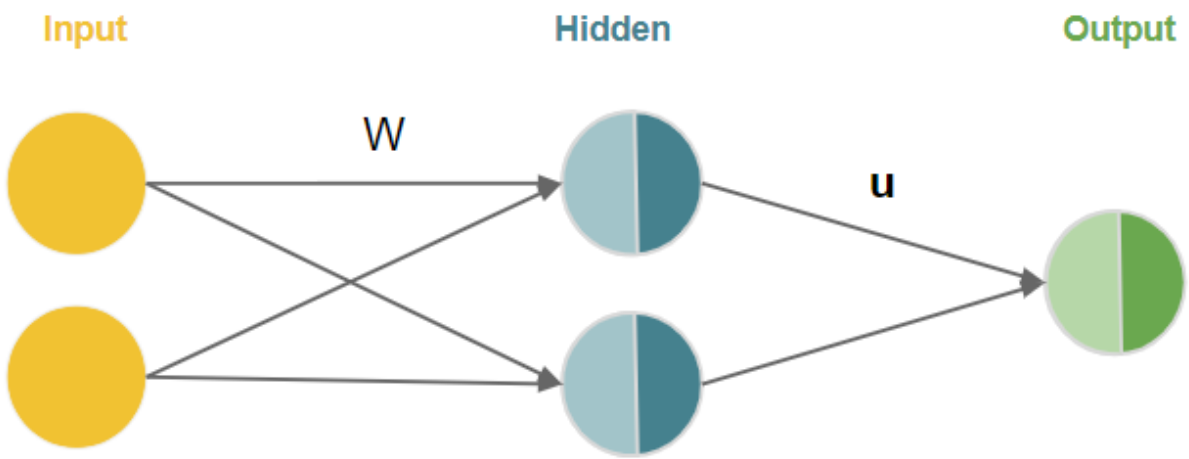
- As activations (hidden and outputs) we use the logistic.
- We run one FP and BP on $\mathbf{x} = (1, 0)^T$ with $y = 1$.
- We use L2 loss between 0-1 labels and the predicted probabilities. This is a bit uncommon, but computations become simpler.



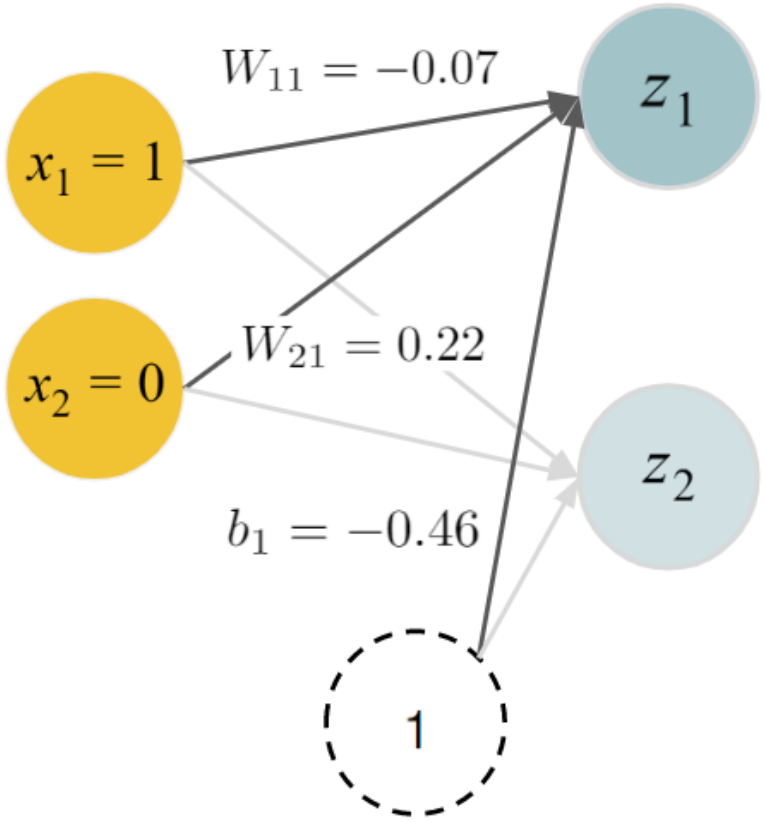
Note: We will only show rounded decimals.

FORWARD PASS

- We will divide the FP into four steps:
  - the inputs of z_i: $z_{i,in}$
  - the activations of z_i: $z_{i,out}$
  - the input of f: f_{in}
  - and finally the activation of f: f_{out}



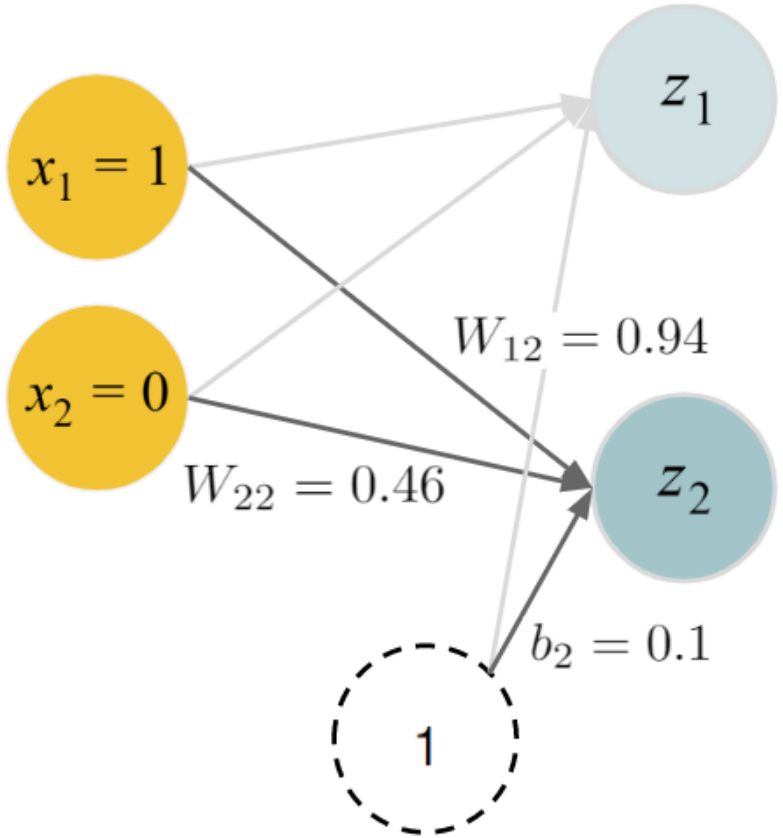
FORWARD PASS



$$z_{1,in} = \mathbf{W}_1^T \mathbf{x} + b_1 = 1 \cdot (-0.07) + 0 \cdot 0.22 + 1 \cdot (-0.46) = -0.53$$

$$z_{1,out} = \sigma(z_{1,in}) = \frac{1}{1 + \exp(-(-0.53))} = 0.3705$$

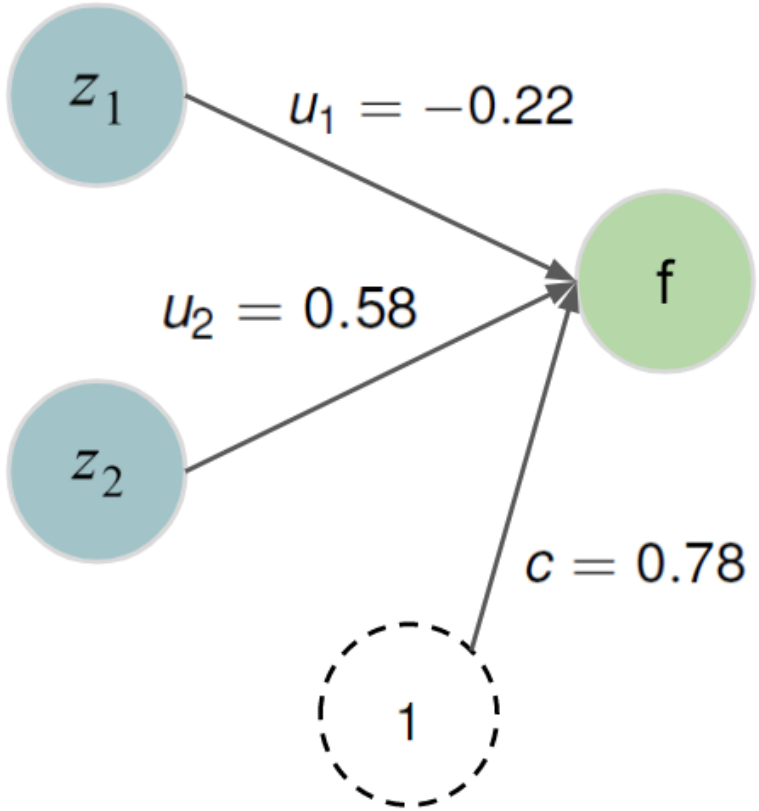
FORWARD PASS



$$z_{2,in} = \mathbf{W}_2^T \mathbf{x} + b_2 = 1 \cdot 0.94 + 0 \cdot 0.46 + 1 \cdot 0.1 = 1.04$$

$$z_{2,out} = \sigma\left(z_{2,in}\right) = \frac{1}{1 + \exp(-1.04)} = 0.7389$$

FORWARD PASS



$$f_{in} = \boldsymbol{u}^T \boldsymbol{z} + c = 0.3705 \cdot (-0.22) + 0.7389 \cdot 0.58 + 1 \cdot 0.78 = 1.1122$$

$$f_{out} = \tau(f_{in}) = \frac{1}{1 + \exp(-1.1122)} = 0.7525$$

FORWARD PASS

- The FP predicted $f_{out} = 0.7525$
- Now, we compare the prediction $f_{out} = 0.7525$ and the true label $y = 1$ using the L2-loss:

$$\begin{aligned} L(y, f(\mathbf{x})) &= \frac{1}{2}(y - f\left(\mathbf{x}^{(i)} \mid \boldsymbol{\theta}\right))^2 = \frac{1}{2}(y - f_{out})^2 \\ &= \frac{1}{2}(1 - 0.7525)^2 = 0.0306 \end{aligned}$$

- The calculation of the gradient is performed backwards (starting from the output layer), so that results can be reused.

BACKWARD PASS

The main ingredients of the backward pass are:

- to reuse the results of the forward pass
  (here: $z_{i,in}, z_{i,out}, f_{in}, f_{out}$)
- reuse the intermediate results from the chain rule
- the derivative of the activations and some affine functions

BACKWARD PASS

- Let's start to update u_1. We recursively apply the chain rule:

$$\frac{\partial L(y, f(\mathbf{x}))}{\partial u_1} = \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial u_1}$$

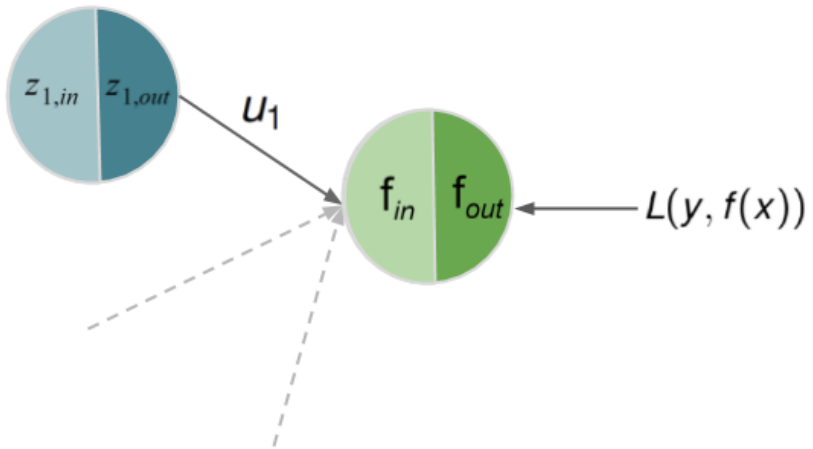
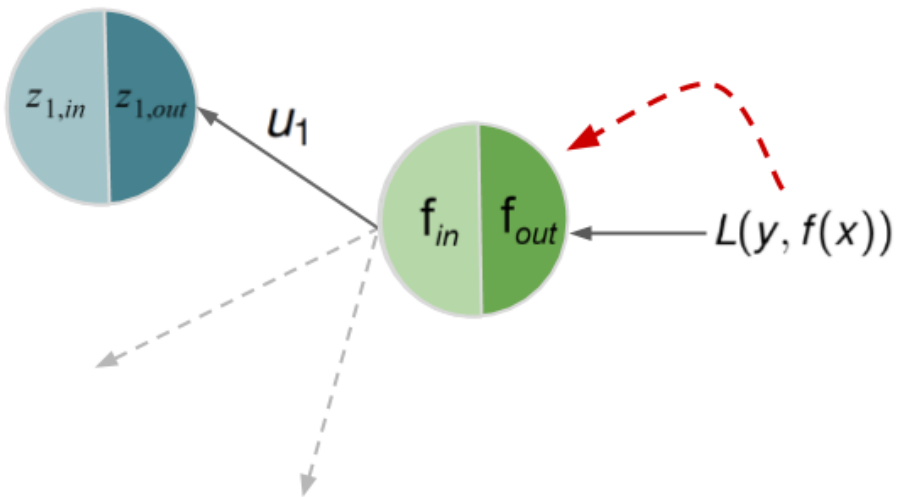


Figure: Snippet from our NN, with backward path for u_1.

BACKWARD PASS

- 1st step: The derivative of L2 is easy; we know f_{out} from FP.

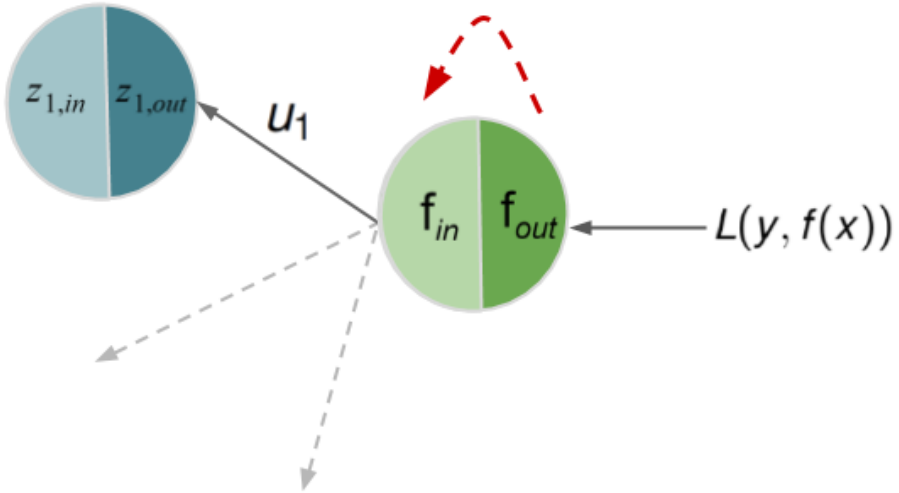
$$\frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} = \frac{d}{\partial f_{out}} \frac{1}{2}(y - f_{out})^2 = -\underbrace{(y - f_{out})}_{\hat{=}\text{residual}}$$
$$= -(1 - 0.7525) = -0.2475$$



BACKWARD PASS

- 2nd step. $f_{out} = \sigma(f_{in})$, use rule for σ', use f_{in} from FP.

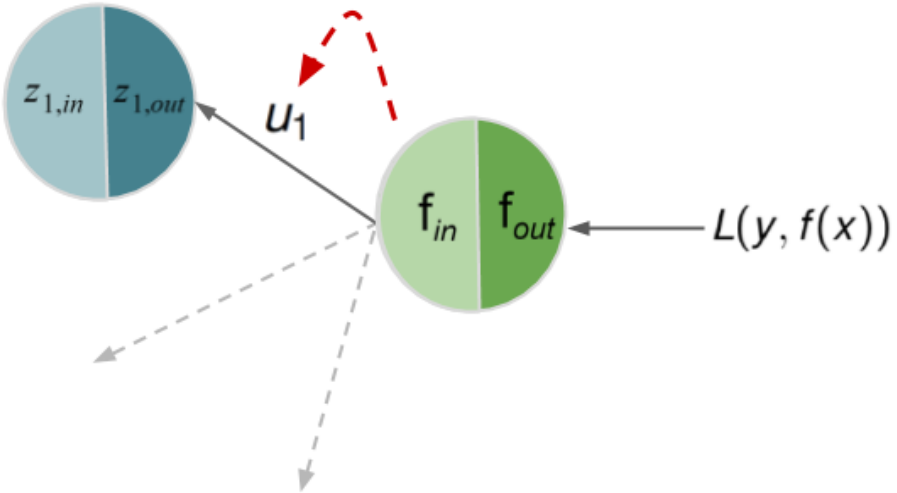
$$\begin{aligned} \frac{\partial f_{out}}{\partial f_{in}} &= \sigma(f_{in}) \cdot (1 - \sigma(f_{in})) \\ &= 0.7525 \cdot (1 - 0.7525) = 0.1862 \end{aligned}$$



BACKWARD PASS

- 3rd step. Derivative of the linear input is easy; use $z_{1,out}$ from FP.

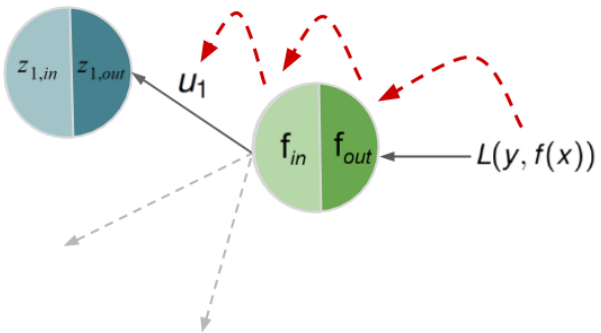
$$\frac{\partial f_{in}}{\partial u_1} = \frac{\partial(u_1 \cdot z_{1,out} + u_2 \cdot z_{2,out} + c \cdot 1)}{\partial u_1} = z_{1,out} = 0.3705$$



BACKWARD PASS

- Plug it together:

$$\begin{aligned}\frac{\partial L(y, f(\mathbf{x}))}{\partial u_1} &= \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial u_1} \\ &= -0.2475 \cdot 0.1862 \cdot 0.3705 = -0.0171\end{aligned}$$



- With LR $\alpha = 0.5$:

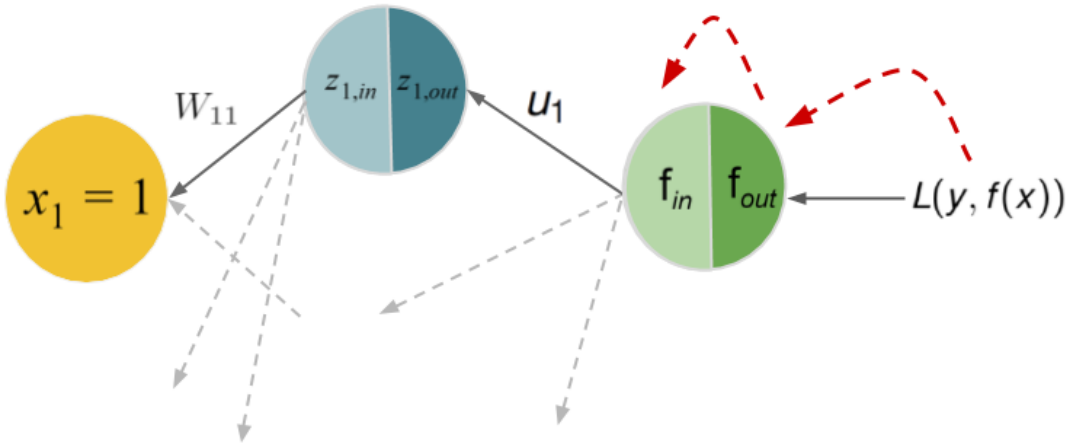
$$\begin{aligned}u_1^{[new]} &= u_1^{[old]} - \alpha \cdot \frac{\partial L(y, f(\mathbf{x}))}{\partial u_1} \\ &= -0.22 - 0.5 \cdot (-0.0171) = -0.2115\end{aligned}$$

BACKWARD PASS

- Now for W_{11}:

$$\frac{\partial L(y, f(\mathbf{x}))}{\partial W_{11}} = \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial z_{1,out}} \cdot \frac{\partial z_{1,out}}{\partial z_{1,in}} \cdot \frac{\partial z_{1,in}}{\partial W_{11}}$$

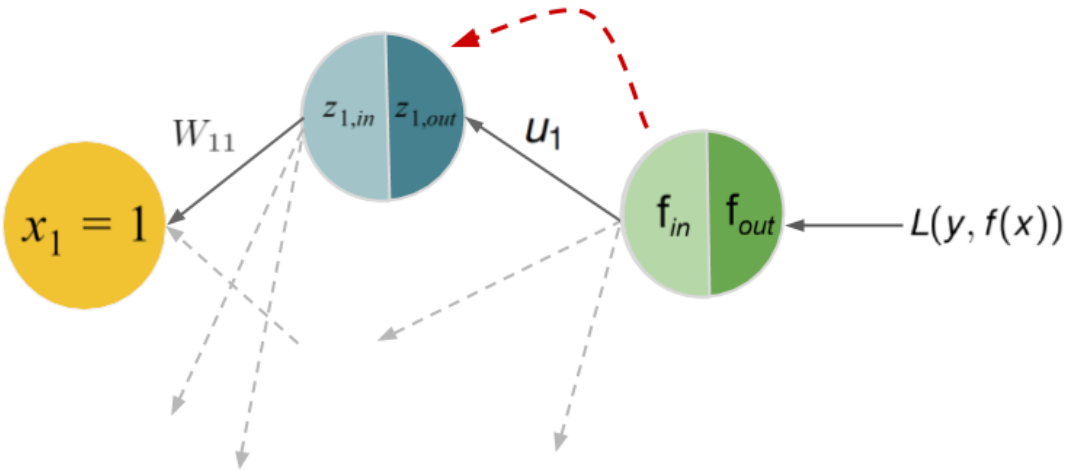
- We know $\frac{\partial L(y,f(\mathbf{x}))}{\partial f_{out}}$ and $\frac{\partial f_{out}}{\partial f_{in}}$ from BP for u_1.



BACKWARD PASS

- $f_{in} = u_1 \cdot z_{1,out} + u_2 \cdot z_{2,out} + c \cdot 1$ is linear, easy and we know u_1 :

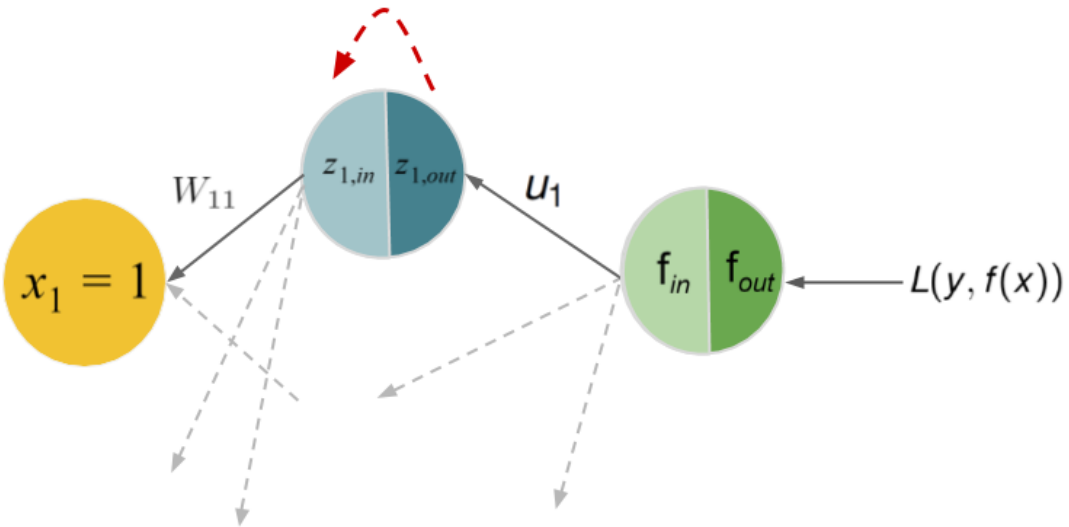
$$\frac{\partial f_{in}}{\partial z_{1,out}} = u_1 = -0.22$$



BACKWARD PASS

- Next. Use rule for σ' and FP results:

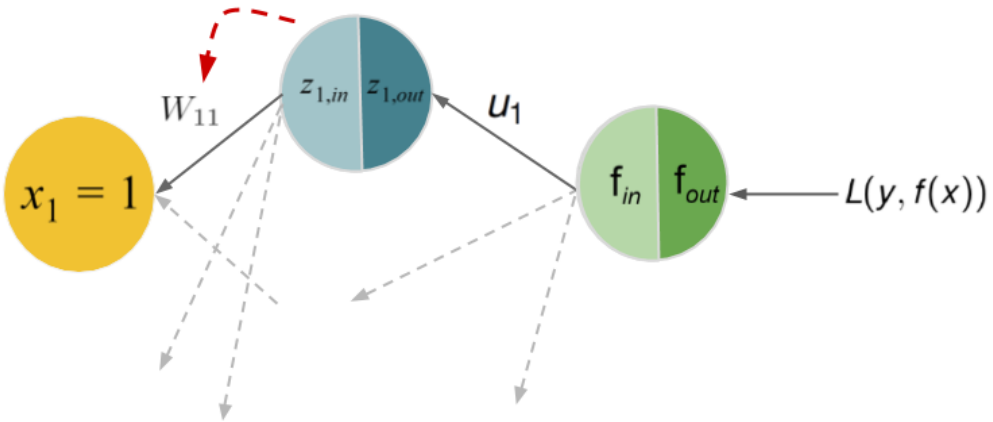
$$\begin{aligned} \frac{\partial z_{1,out}}{\partial z_{1,in}} &= \sigma(z_{1,in}) \cdot (1 - \sigma(z_{1,in})) \\ &= 0.3705 \cdot (1 - 0.3705) = 0.2332 \end{aligned}$$



BACKWARD PASS

- $z_{1,in} = x_1 \cdot W_{11} + x_2 \cdot W_{21} + b_1 \cdot 1$ is linear and depends on inputs:

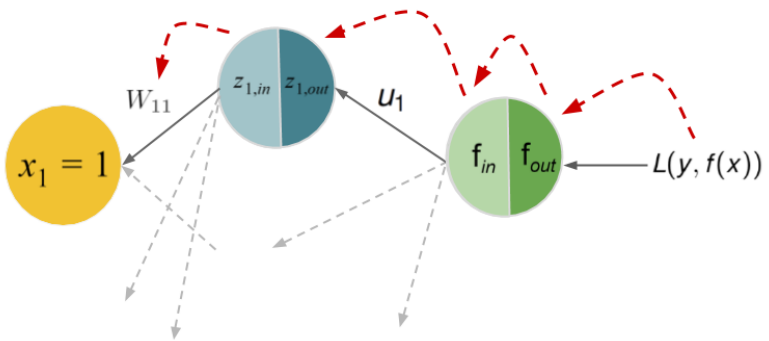
$$\frac{\partial z_{1,in}}{\partial W_{11}} = x_1 = 1$$



BACKWARD PASS

- Plugging together:

$$\begin{aligned}\frac{\partial L(y, f(\mathbf{x}))}{\partial W_{11}} &= \frac{\partial L(y, f(\mathbf{x}))}{\partial f_{out}} \cdot \frac{\partial f_{out}}{\partial f_{in}} \cdot \frac{\partial f_{in}}{\partial z_{1,out}} \cdot \frac{\partial z_{1,out}}{\partial z_{1,in}} \cdot \frac{\partial z_{1,in}}{\partial W_{11}} \\ &= (-0.2475) \cdot 0.1862 \cdot (-0.22) \cdot 0.2332 \cdot 1 \\ &= 0.0024\end{aligned}$$



- Full SGD update:

$$\begin{aligned}W_{11}^{[new]} &= W_{11}^{[old]} - \alpha \cdot \frac{\partial L(y, f(\mathbf{x}))}{\partial W_{11}} \\ &= -0.07 - 0.5 \cdot 0.0024 = -0.0712\end{aligned}$$

RESULT

- We can do this for all weights:

$$W = \begin{pmatrix} -0.0712 & 0.9426 \\ 0.22 & 0.46 \end{pmatrix}, b = \begin{pmatrix} -0.4612 \\ 0.1026 \end{pmatrix},$$

$$u = \begin{pmatrix} -0.2115 \\ 0.5970 \end{pmatrix} \text{ and } c = 0.8030.$$

- Yields $f(\mathbf{x} \mid \boldsymbol{\theta}^{[new]}) = 0.7615$ and loss $\frac{1}{2}(1 - 0.7615)^2 = 0.0284$.
- Before, we had $f(\mathbf{x} \mid \boldsymbol{\theta}^{[old]}) = 0.7525$ and higher loss 0.0306.

Now rinse and repeat. This was one training iter, we do thousands.

DEEP NEURAL NETWORKS

Neural networks today can have hundreds of hidden layers. The greater the number of layers, the "deeper" the network. Historically DNNs were very challenging to train and not popular until the late ’00s for several reasons:

- The use of sigmoid activations (e.g., logistic sigmoid and tanh) significantly slowed down training due to a phenomenon known as “vanishing gradients”. The introduction of the ReLU activation largely solved this problem.
- Training DNNs on CPUs was too slow to be practical. Switching over to GPUs cut down training time by more than an order of magnitude.
- When dataset sizes are small, other models (such as SVMs) and techniques (such as feature engineering) often outperform them.

DEEP NEURAL NETWORKS

- The availability of large datasets and novel architectures that are capable of handling even complex tensor-shaped data (e.g. CNNs for image data), faster hardware, and better optimization and regularization methods made it feasible to successfully implement deep neural networks.

- An increase in depth often translates to an increase in performance on a given task. State-of-the-art neural networks, however, are much more sophisticated than the simple architectures we have encountered so far.

The term "**deep learning**" encompasses all of these developments and refers to the field as a whole.

Where NNs thrive

> **Statistical/correlation inference** needed

> There exists **a lot of good quality (labelled) training data**

> **Parallelizable** training and deployment

> Tasks **without expansion** (input-output fixed)

> **Specialized tasks**

> Good **in-range performance** IRL

https://lasp.colorado.edu/home/minxss/2016/07/12/minimum-mission-success-criteria-met/

Limits of NNs

> **No causal relations** possible (yet)

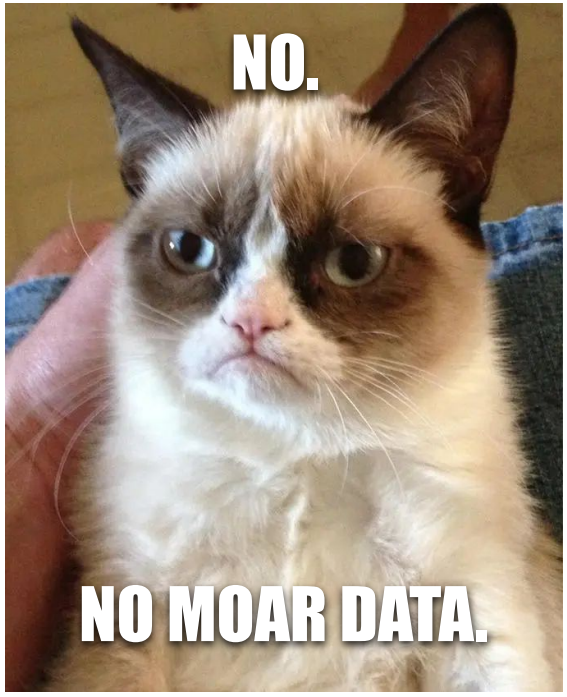
> **Very data hungry** - "*Garbage in, garbage out*"

> **Often expensive to train** (depending on size)

> **Nonextensible** and **specialized** to a range and task

- Add one more neuron → needs fine-tuning
- Undefined behaviour on out-of-domain test examples

Note on specialization: rf. 'Foundation Models'



https://knowyourmeme.com/memes/grumpy-cat

Overfitting & Underfitting

The real troublemakers in ML in general!

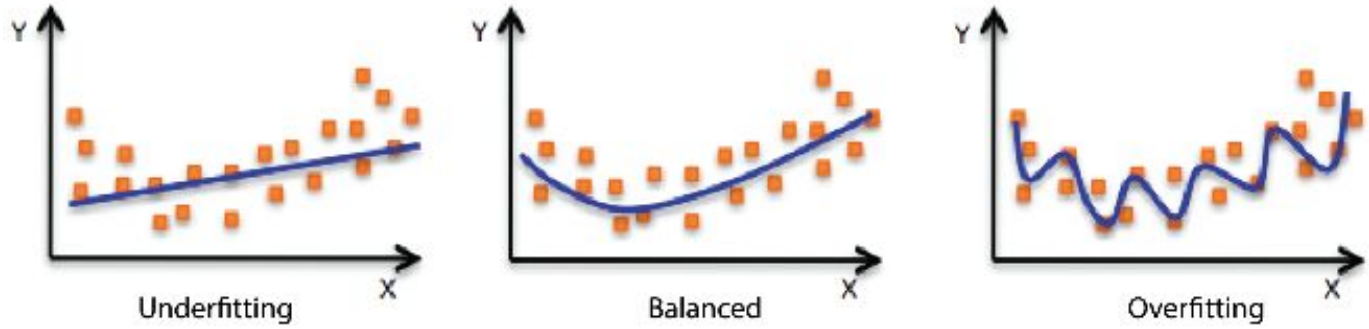
Underfitting: When the model fits the training data not well enough

- Empirical risk is high, actual risk is high
- Training loss is high, testing loss is not optimal

Overfitting: When the model fits the training data *too* closely (incl. noise)

- Empirical risk is low, actual risk is high
- Training loss is low, testing loss is not optimal
- e.g. An *D*-degree polynomial can fit *D-1* training points with *zero* error

Overfitting & Underfitting



More complex models (e.g. more layers, neurons per layer) -> higher likelihood of overfitting

Image source: https://docs.aws.amazon.com/machine-learning/latest/dg/model-fit-underfitting-vs-overfitting.html

Validation

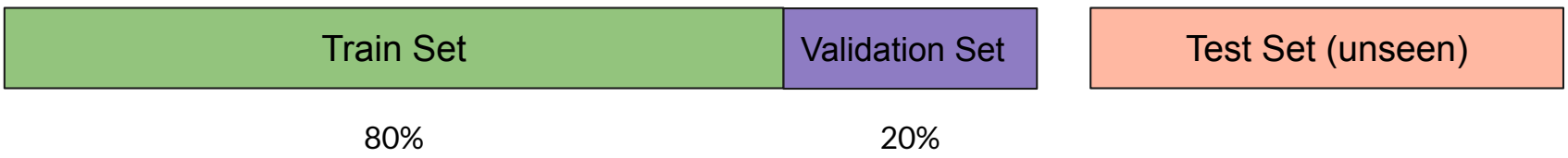
Split your training set into two!

- New train set
- Unseen-by-the-model “validation”set

Validation

Split your training set into two!

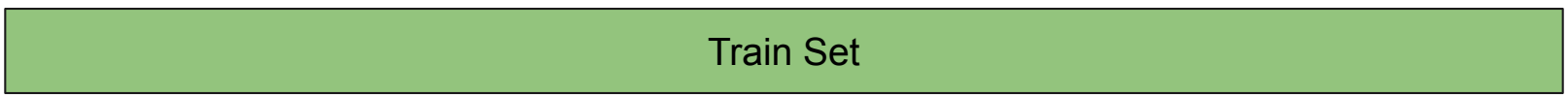
- New train set
- Unseen-by-the-model “validation”set
- e.g. 80-20 split (Note: split ratio depends on the model, task and data)



k-fold Cross-validation

Split training set into k-segments, iteratively train and validate with each segment.

- Accounts for irregularities in training set
- “Gold standard” for evaluating generality of neural network models

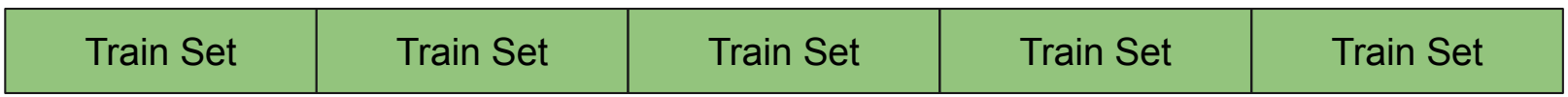


k-fold Cross-validation

Split training set into k-segments, iteratively train and validate with each segment.

- Accounts for irregularities in training set
- “Gold standard” for evaluating generality of neural network models

e.g. *k*=5 (5-fold cross-validation)

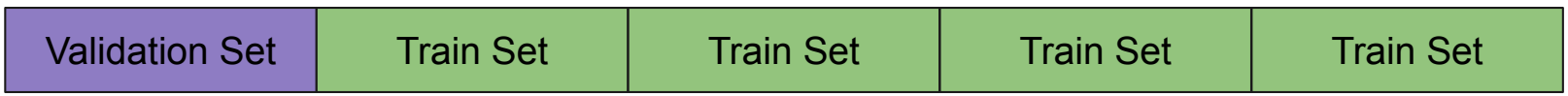


k-fold Cross-validation

Split training set into k-segments, iteratively train and validate with each segment.

- Accounts for irregularities in training set
- “Gold standard” for evaluating generality of neural network models

e.g. *k*=5 (5-fold cross-validation)

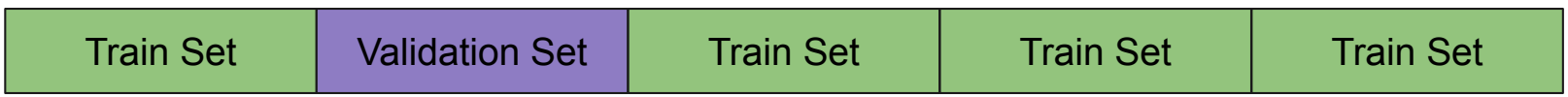


k-fold Cross-validation

Split training set into k-segments, iteratively train and validate with each segment.

- Accounts for irregularities in training set
- “Gold standard” for evaluating generality of neural network models

e.g. *k*=5 (5-fold cross-validation)



k-fold Cross-validation

Split training set into k-segments, iteratively train and validate with each segment.

- Accounts for irregularities in training set
- “Gold standard” for evaluating generality of neural network models

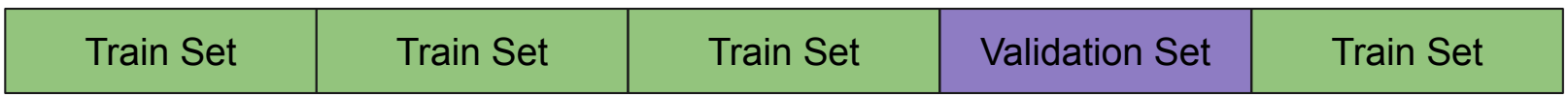
e.g. *k*=5 (5-fold cross-validation)

k-fold Cross-validation

Split training set into k-segments, iteratively train and validate with each segment.

- Accounts for irregularities in training set
- “Gold standard” for evaluating generality of neural network models

e.g. *k*=5 (5-fold cross-validation)

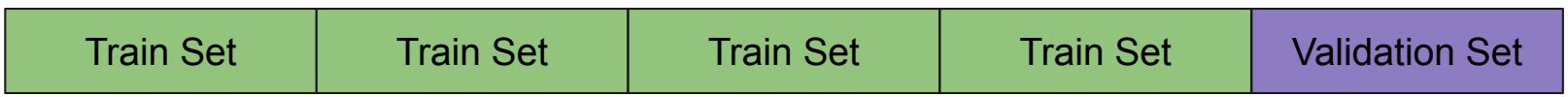


k-fold Cross-validation

Split training set into k-segments, iteratively train and validate with each segment.

- Accounts for irregularities in training set
- “Gold standard” for evaluating generality of neural network models

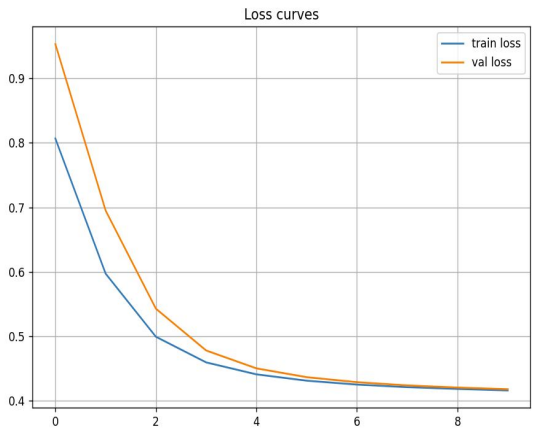
e.g. *k*=5 (5-fold cross-validation)

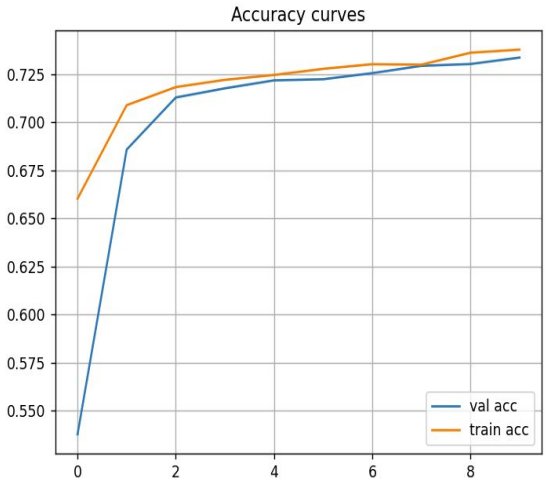


Result = average over all validation passes

Training curves

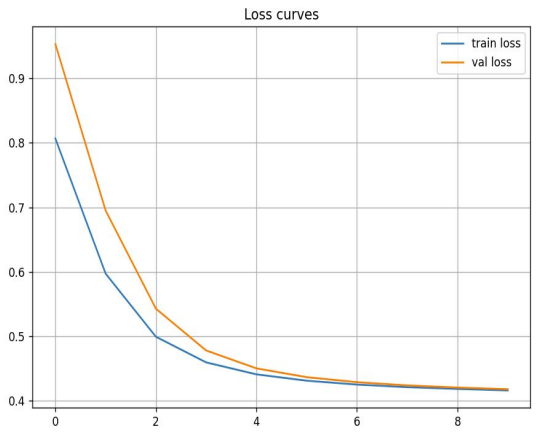
Important to plot!

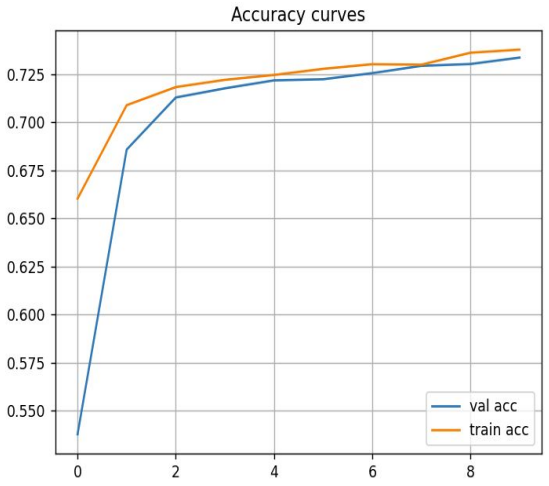




Training curves

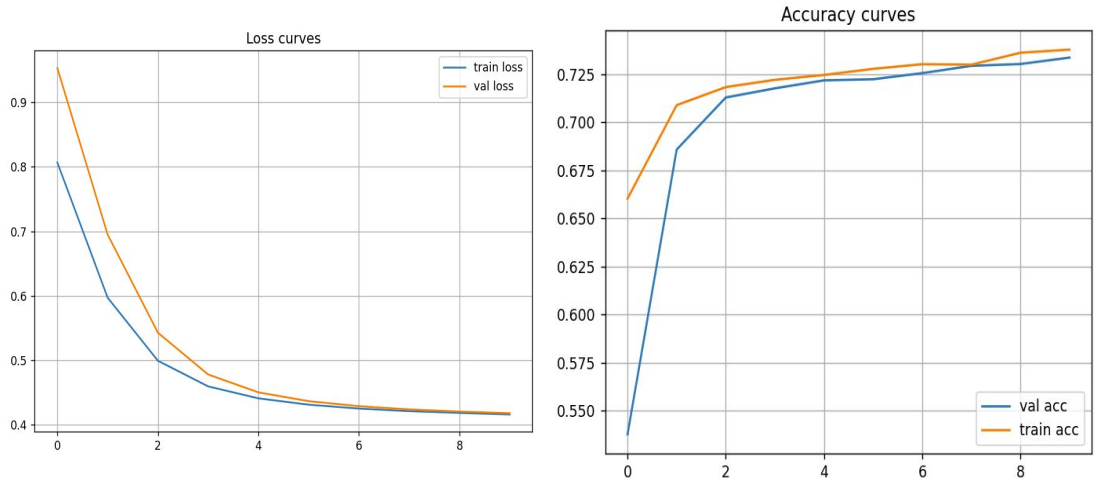
Important to plot!(!!!!)





Training curves

Important to plot!(!!!!)

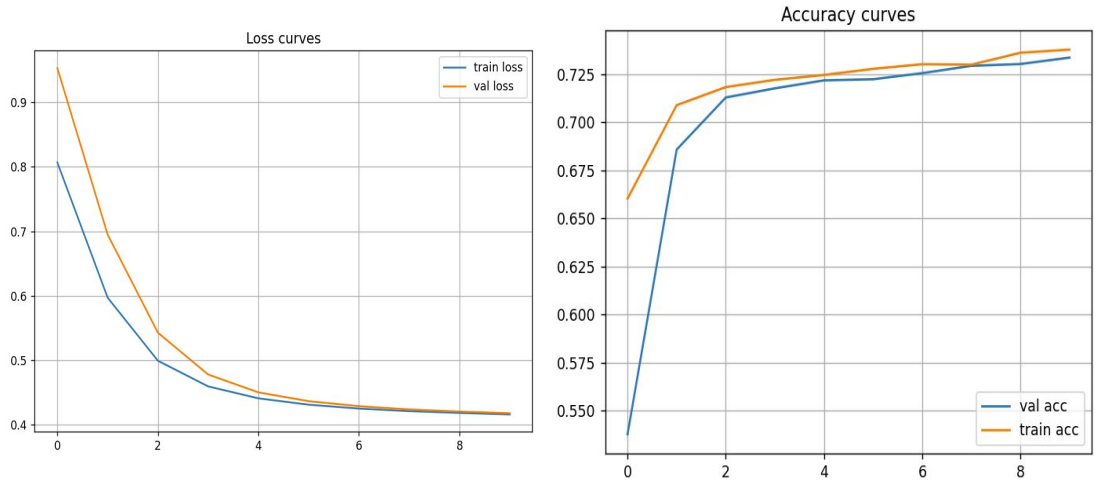


Shows if and how fast your model is learning on task-relevant metrics

- e.g. loss, accuracy, AUC, F1 score
- Plot scores over training epochs

Training curves

Important to plot!(!!!!)

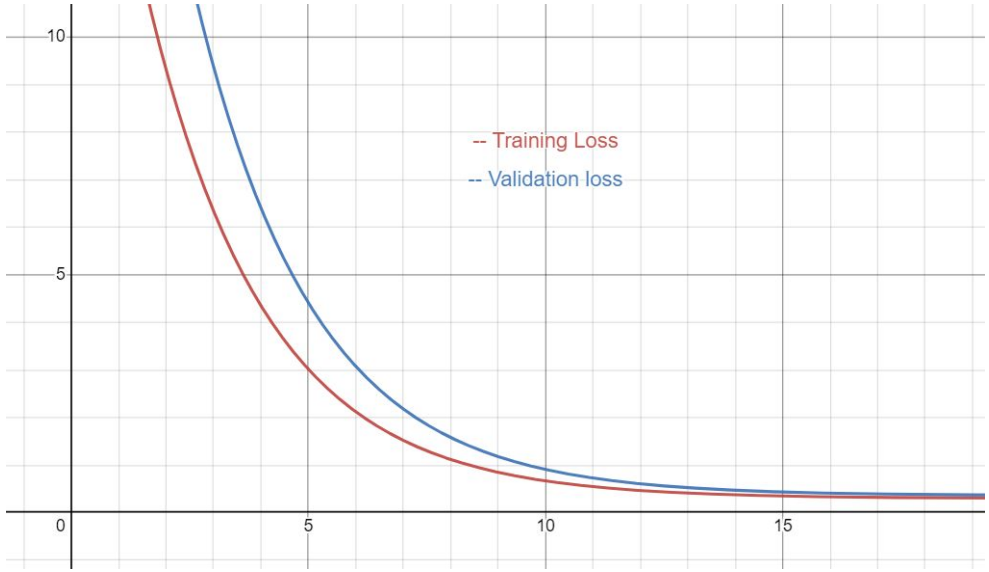


Shows if and how fast your model is learning on task-relevant metrics

- e.g. loss, accuracy, AUC, F1 score
- Plot scores over training epochs

May indicate potential over and underfitting

Reading training curves



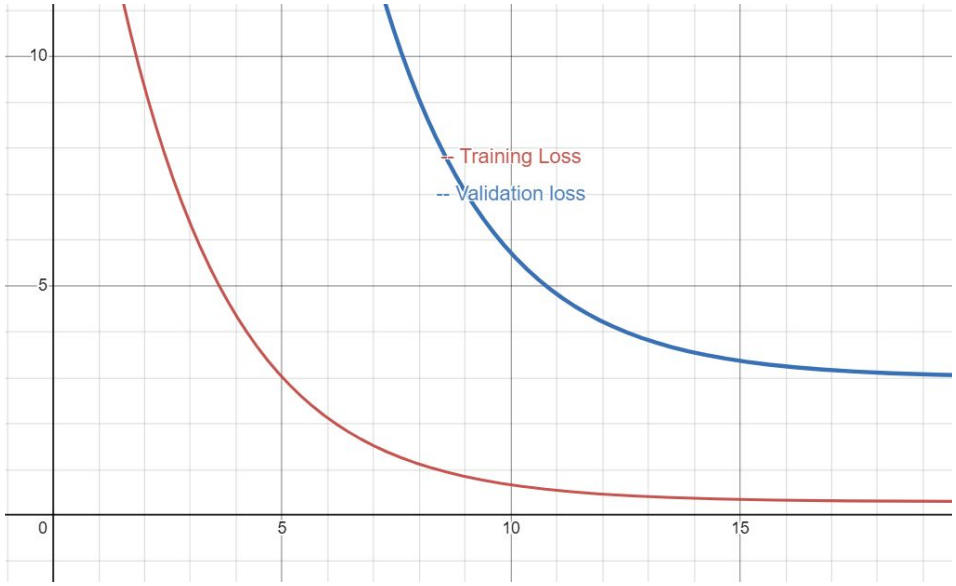
If

validation loss > training loss

then often the model is *good!*

Low loss == Better

Reading training curves



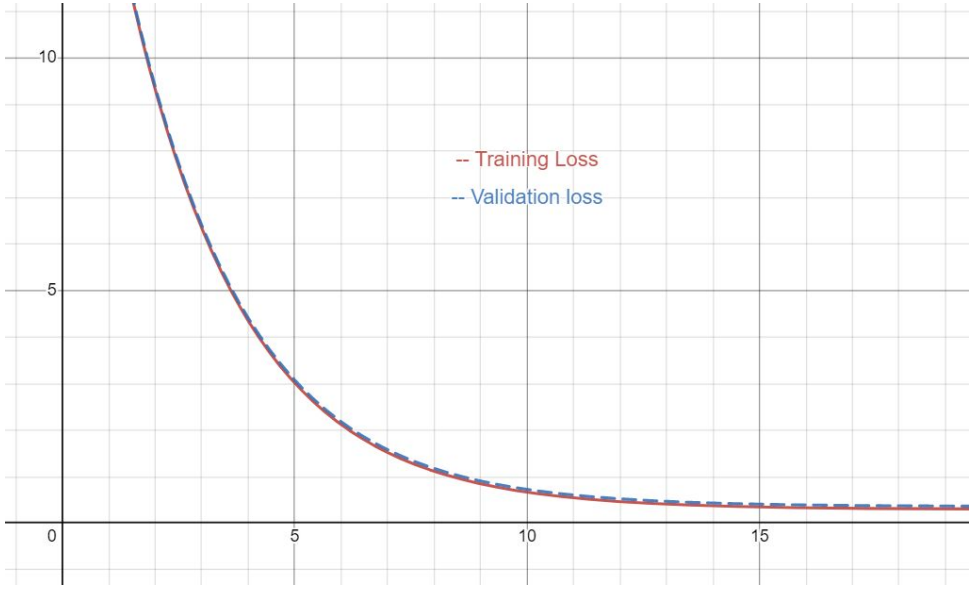
If

validation loss >> training loss

then often the model is *overfitting*

Low loss == Better

Reading training curves



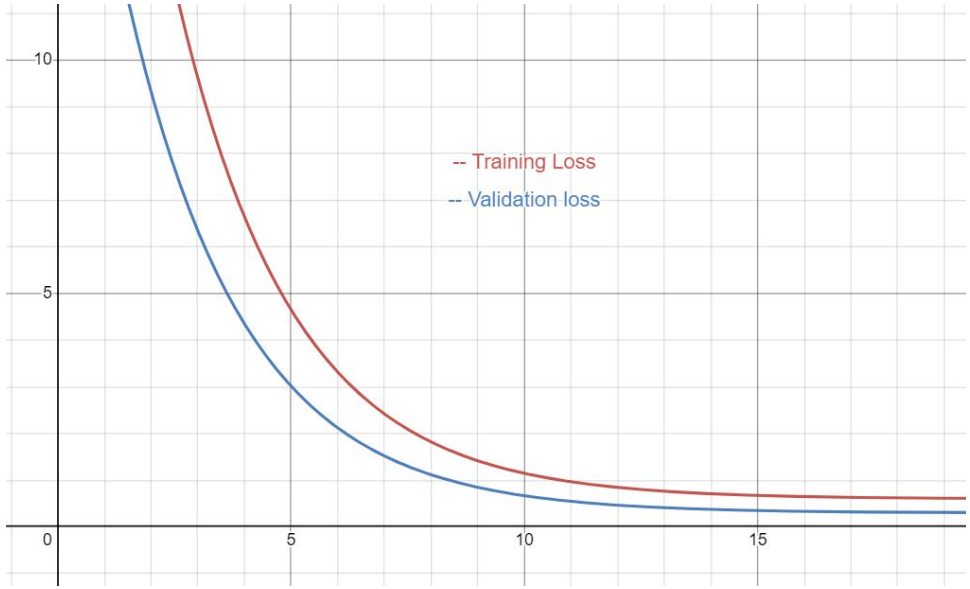
If

validation loss ~ training loss

then often the model is *underfitting*

Low loss == Better

Reading training curves



If

validation loss < training loss

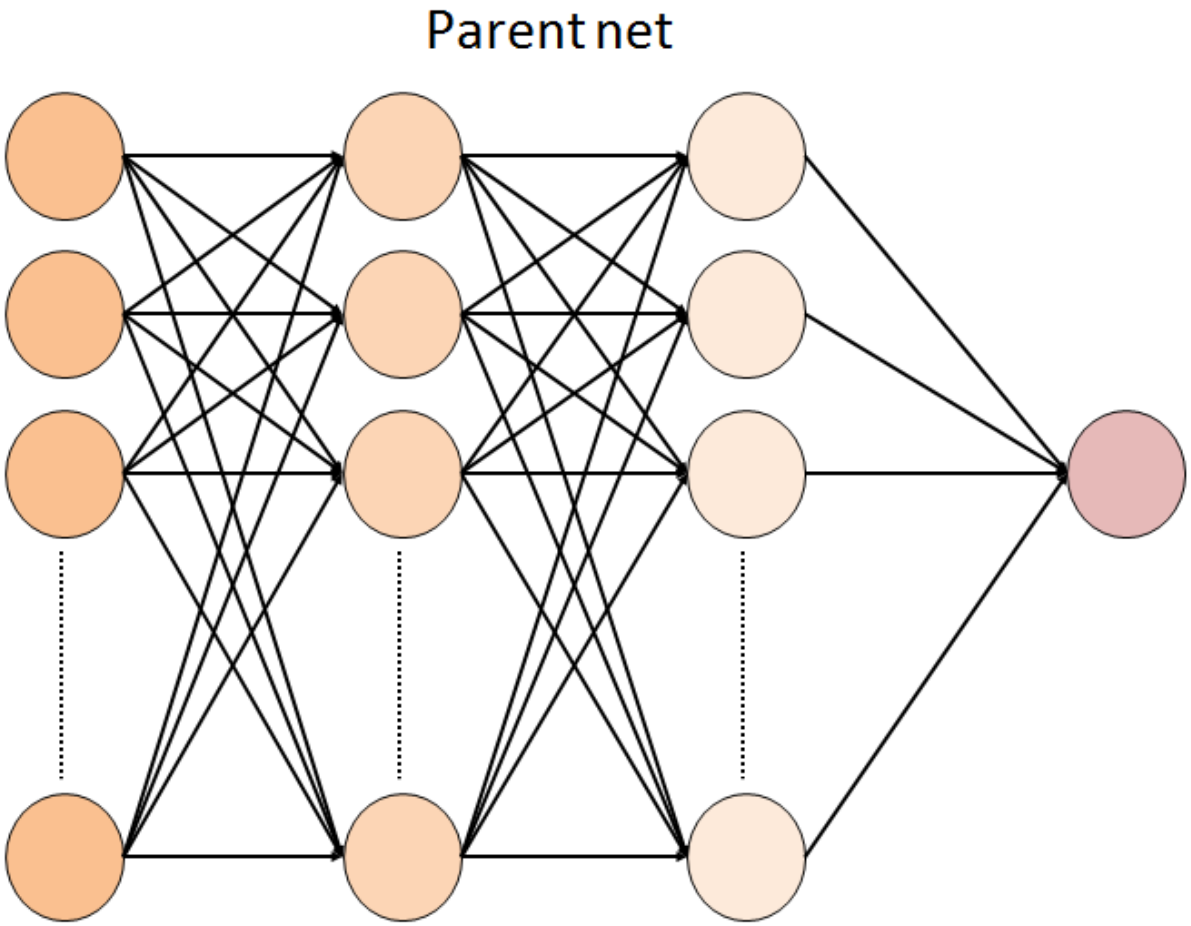
then something is *very* wrong, or *totally expected*!

Low loss == Better

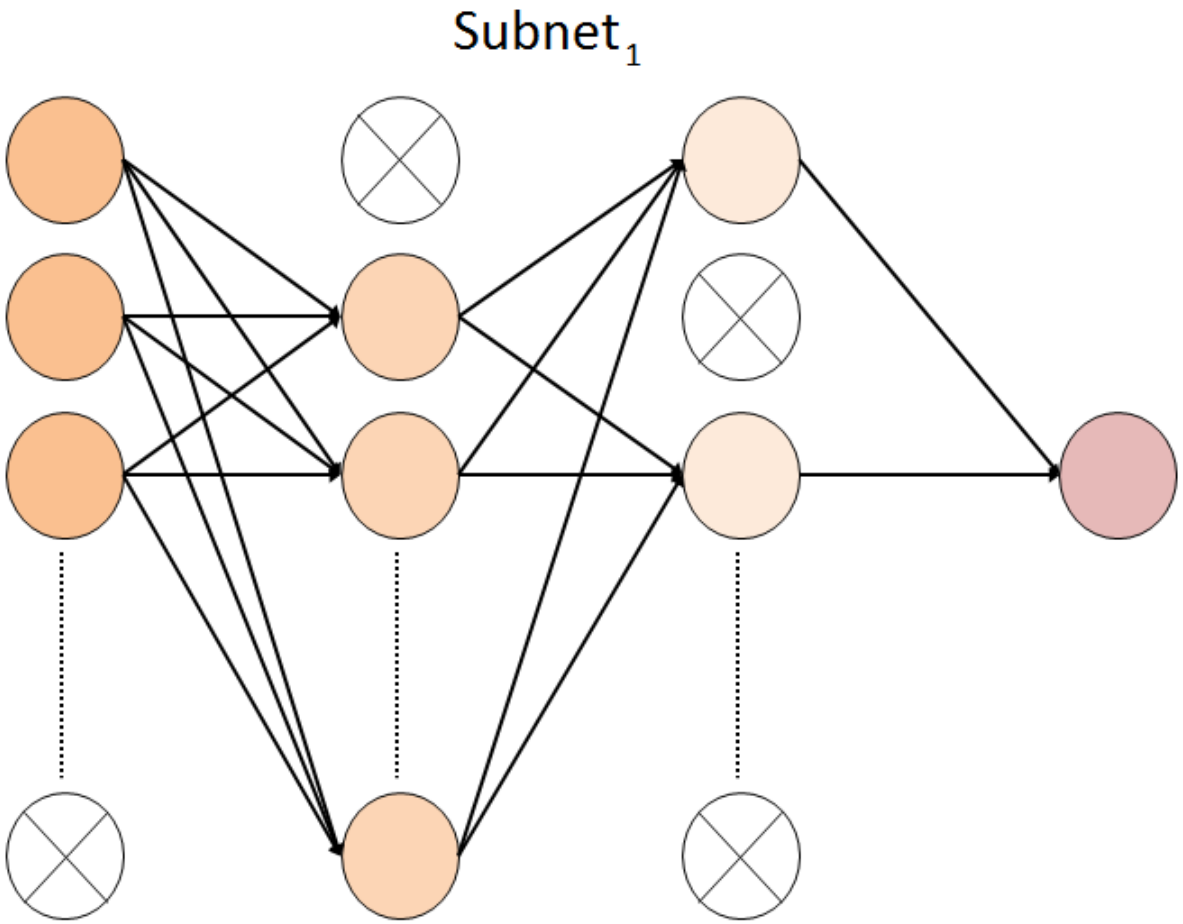
DROPOUT

- Idea: reduce overfitting in neural networks by preventing complex co-adaptations of neurons.
- Method: during training, random subsets of the neurons are removed from the network (they are "dropped out").This is done by artificially setting the activations of those neurons to zero.
- Whether a given unit/neuron is dropped out or not is completely independent of the other units.
- If the network has N (input/hidden) units, applying dropout to these units can result in 2^N possible 'subnetworks'.
- Because these subnetworks are derived from the same 'parent' network, many of the weights are shared.
- Dropout can be seen as a form of "model averaging".

DROPOUT

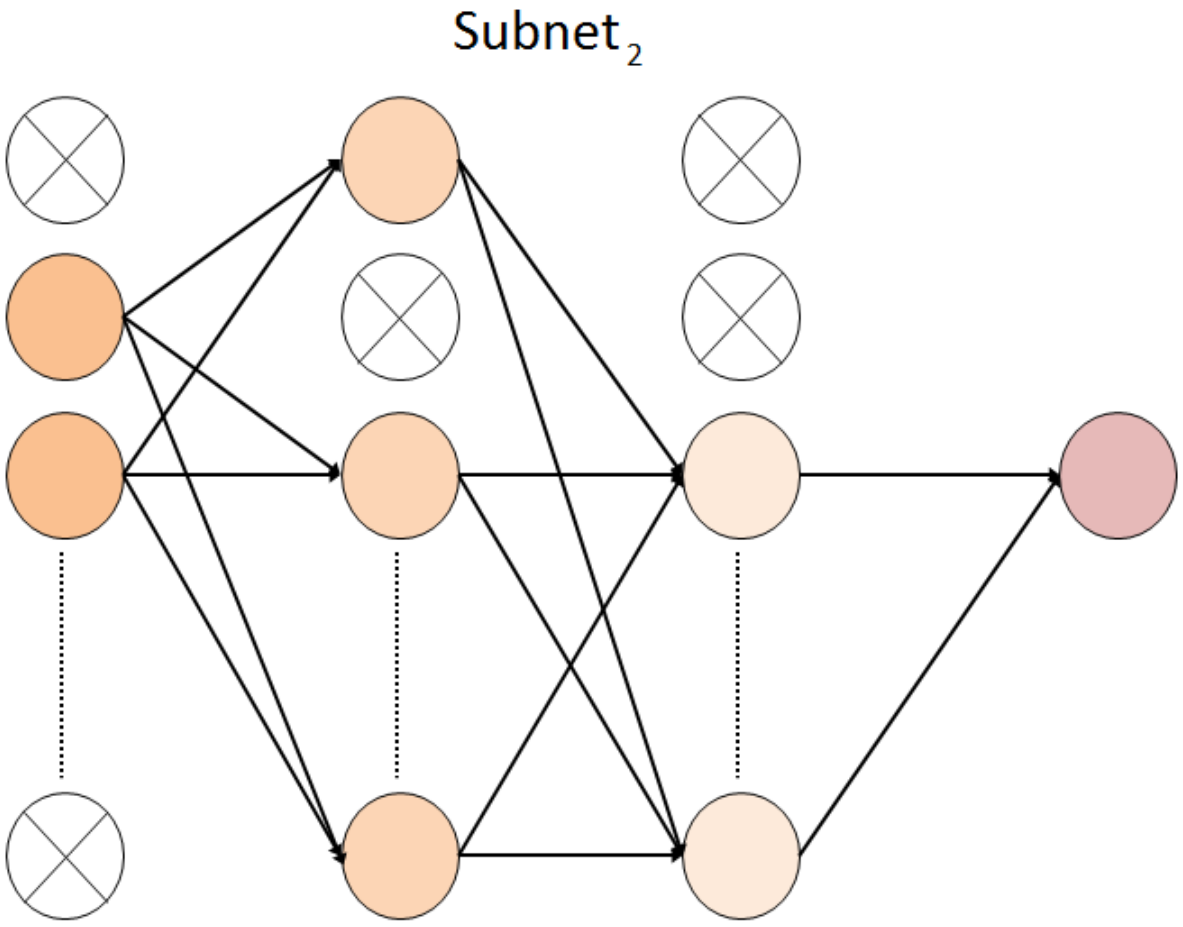


DROPOUT



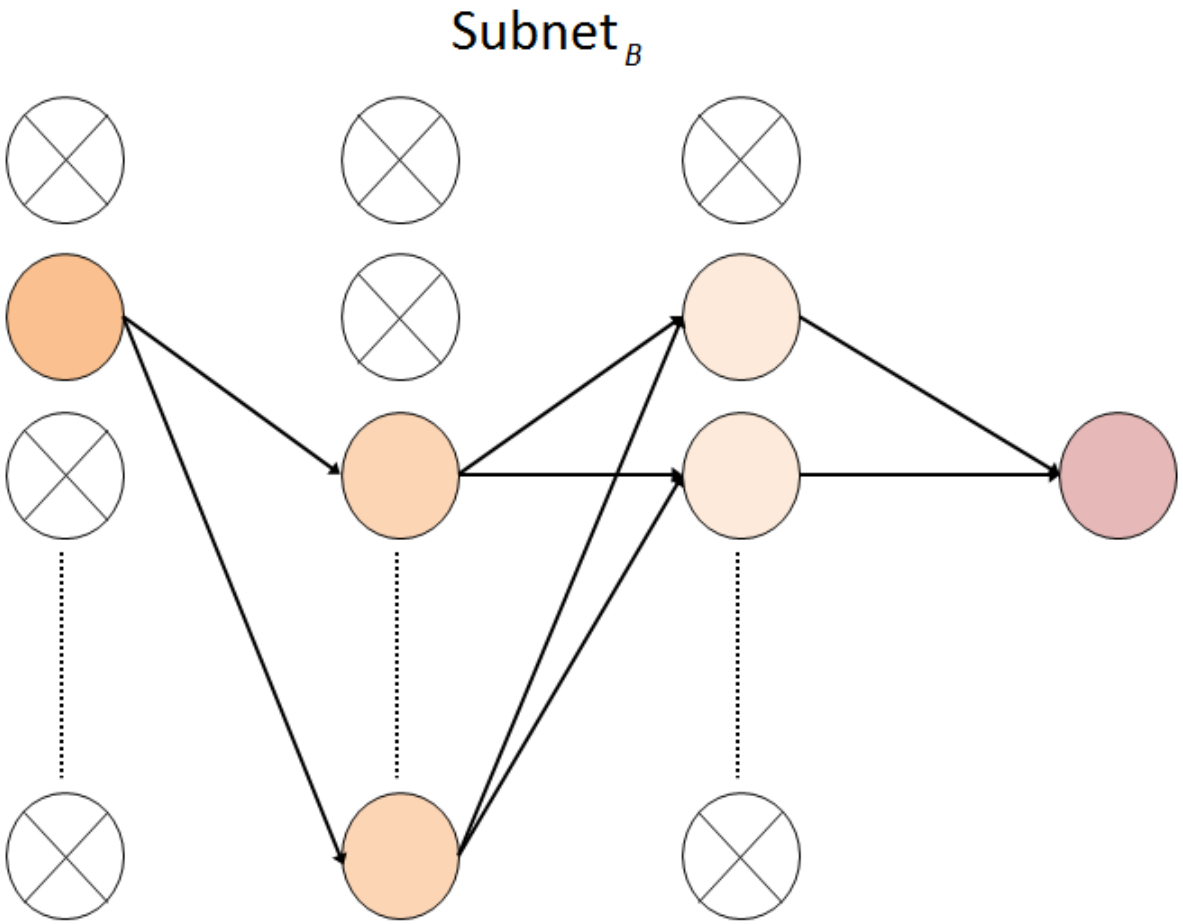
In each iteration, for each training example (in the forward pass), a different (random) subset of neurons are dropped out.

DROPOUT



In each iteration, for each training example (in the forward pass), a different (random) subset of neurons are dropped out.

DROPOUT



In each iteration, for each training example (in the forward pass), a different (random) subset of neurons are dropped out.

DROPOUT: ALGORITHM

- To train with dropout a minibatch-based learning algorithm such as stochastic gradient descent is used.
- For each training case in a minibatch, we randomly sample a binary vector/mask μ with one entry for each input or hidden unit in the network. The entries of μ are sampled independently from each other.
- The probability *p* of sampling a mask value of 0 (dropout) for one unit is a hyperparameter known as the 'dropout rate'.
- A typical value for the dropout rate is 0.2 for input units and 0.5 for hidden units.
- Each unit in the network is multiplied by the corresponding mask value resulting in a $subnet_{\mu}$.
- Forward propagation, backpropagation, and the learning update are run as usual.

DROPOUT: WEIGHT SCALING

- The weights of the network will be larger than normal because of dropout. Therefore, to obtain a prediction at test time the weights must be first scaled by the chosen dropout rate.
- This means that if a unit (neuron) is retrained with probability p during training, the weight at test time of that unit is multiplied by p.

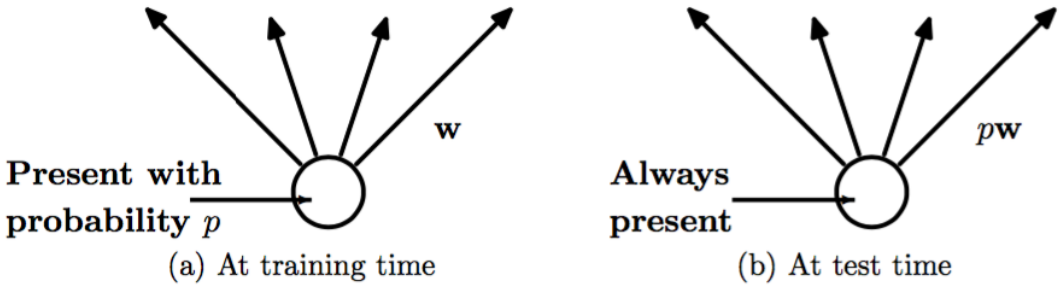
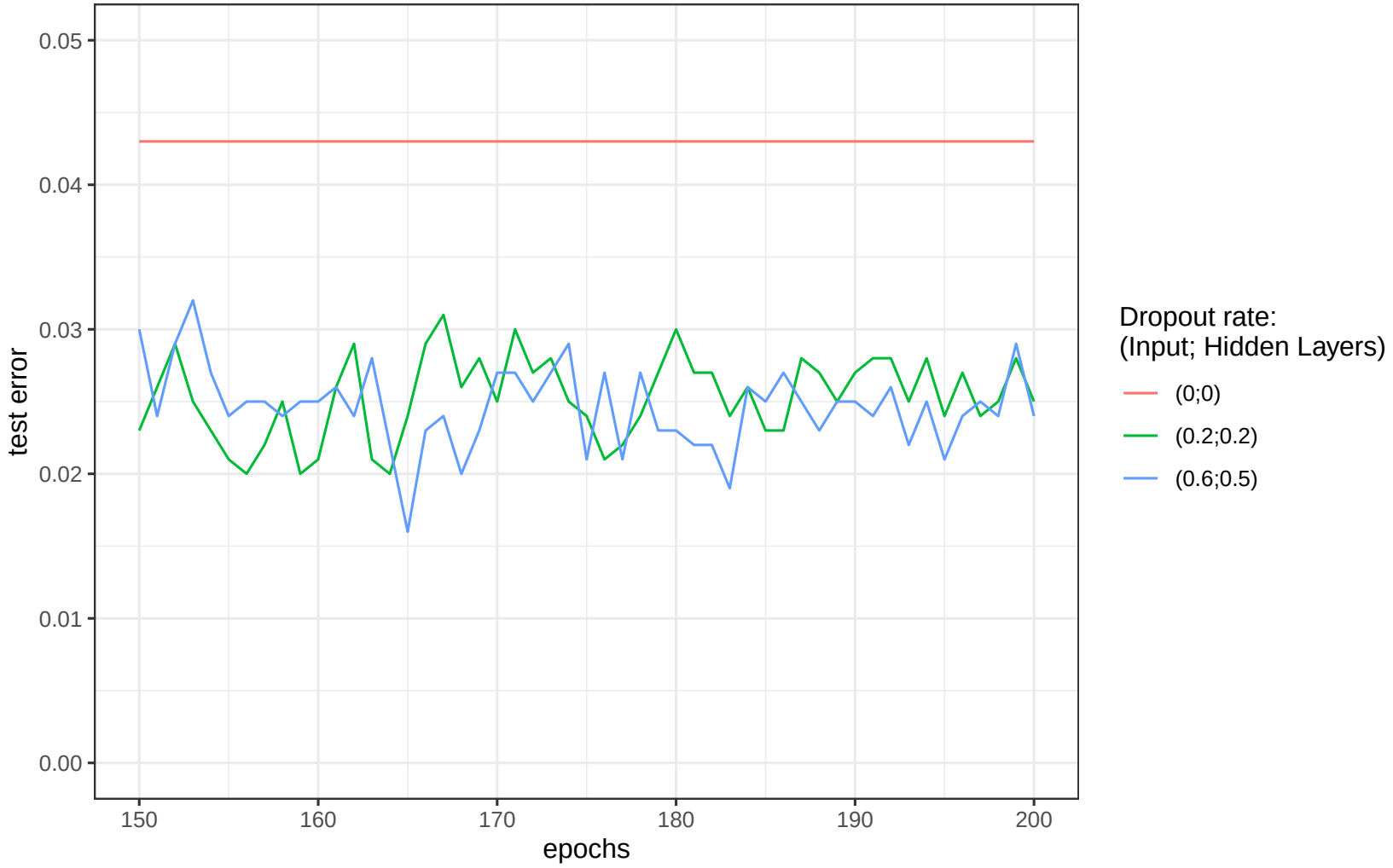


Figure: Training vs. Testing (Srivastava et. al., 2014)

- Weight scaling ensures that the expected total input to a neuron/unit at test time is roughly the same as the expected total input to that unit at train time, even though many of the units at train time were missing on average

DROPOUT: EXAMPLE



Dropout rate of 0 (no dropouts) leads to higher test error than dropping some units out.

DATASET AUGMENTATION

- Problem: low generalization because high ratio of

$$\frac{\text{complexity of the model}}{\text{\#train data}}$$

- Idea: artificially increase the train data.
  - Limited data supply $\rightarrow$ create “fake data”!
- Increase variation in inputs **without** changing the labels.
- Application:
  - Image and Object recognition (rotation, scaling, pixel translation, flipping, noise injection, vignetting, color casting, lens distortion, injection of random negatives)
  - Speech recognition (speed augmentation, vocal tract perturbation)

DATASET AUGMENTATION

(a) Original **(b)** Color

(c) Rotate **(d)** Horizontal Stretch

Figure: Example for data set augmentation (Wu et al., 2015)

DATASET AUGMENTATION

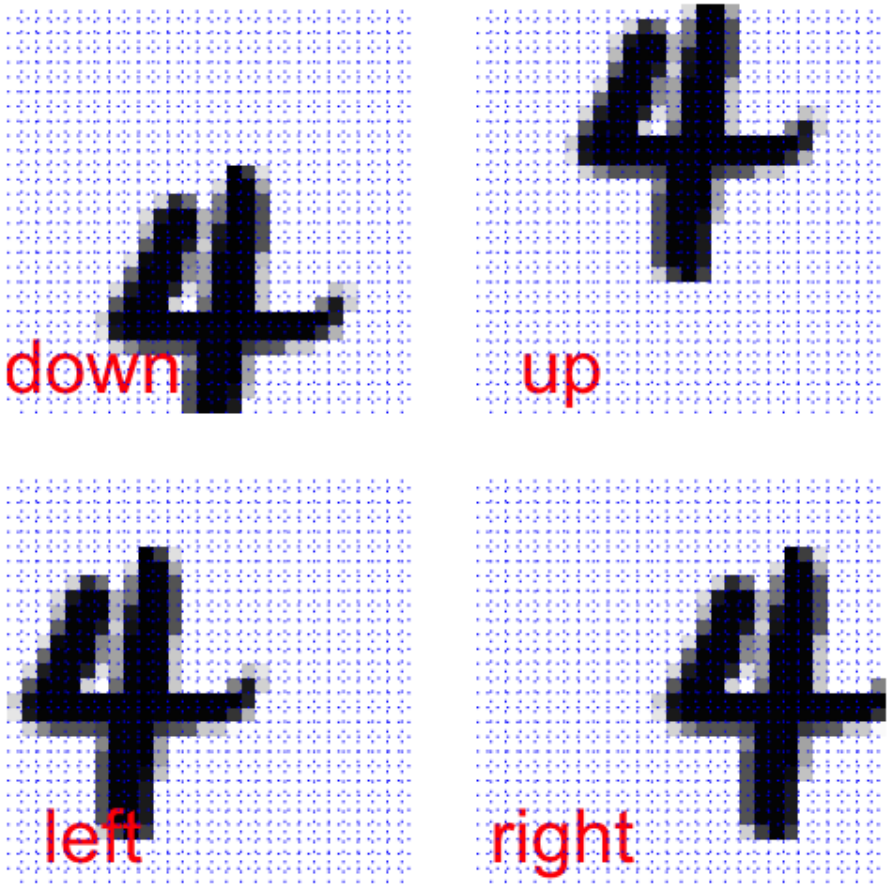


Figure: Example for data set augmentation (Wu et al., 2015)

$\Rightarrow$ careful when rotating digits (6 will become 9 and vice versa)!

Practical advice

Practical advice (1/3)

Hyperparameter	**Underfitting (Symptoms)**	**Overfitting (Symptoms)**	**Recommended Adjustments**
Model Complexity	Too few layers/neurons; high bias	Excessive layers/neurons; memorization of noise	Increase capacity for underfitting; reduce complexity or add regularization for overfitting.
Learning Rate	Too low: slow convergence; too high: unstable updates	—	Increase (or use adaptive optimizers) if learning is too slow; use proper scheduling to maintain stability.
Number of Epochs	Insufficient epochs lead to incomplete learning	Too many epochs lead to memorization	Increase epochs for underfitting; use early stopping to prevent overfitting.
Dropout Rate	Excessively high dropout removes too much information	Insufficient dropout allows over-reliance on specific neurons	Lower dropout for underfitting; increase dropout (e.g., 20–50%) for overfitting.
Reg. (L1/L2)	Overly strong regularization penalizes weights too much	Too weak regularization leads to fitting noise	Decrease regularization strength for underfitting; increase it for overfitting.

Practical advice (2/3)

Hyperparameter	**Underfitting (Symptoms)**	**Overfitting (Symptoms)**	**Recommended Adjustments**
Data Augmentation	Often not needed if the model struggles to learn	Lack of augmentation may cause overfitting on limited data	Use more augmentation to diversify data when overfitting is observed.
Early Stopping	—	Delayed stopping allows memorization of training data	Use early stopping with an appropriate patience (e.g., 3–10 epochs) and `min_delta` (e.g., 0.001).
Batch Size	Too large batches may oversmooth gradients	Too small batches introduce high variance	Experiment with moderate batch sizes to balance stability and update frequency.
Optimizer Parameters	Poor settings (e.g., low momentum) slow convergence	—	Tune momentum or β values (for Adam) to enhance convergence.
Learning Rate Scheduling/Decay	Lack of decay hinders fine-tuning in later stages	Constant high rate prevents settling into a stable minimum	Implement decay schedules to reduce the learning rate as training progresses.

Table: Additional hyperparameters and related tuning suggestions.

Practical advice (3/3)

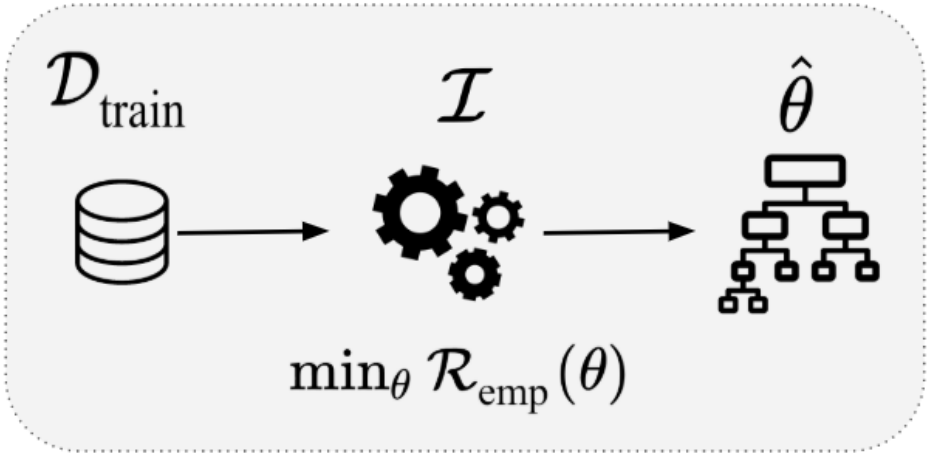
Hyperparameter	**Underfitting (Symptoms)**	**Overfitting (Symptoms)**	**Recommended Adjustments**
Weight Initialization	Inadequate initialization impedes effective learning	—	Use advanced schemes (e.g., He or Glorot initialization) to provide a better starting point.
Activation Functions	Inappropriate choices can lead to vanishing gradients	—	Experiment with activations (e.g., ReLU, Leaky ReLU) to improve gradient flow.
Normalization Techniques	—	Absence of normalization can destabilize training	Apply batch normalization or layer normalization to stabilize and regularize training.

Table: Additional hyperparameters and related tuning suggestions.

Hyperparameter Search Recap

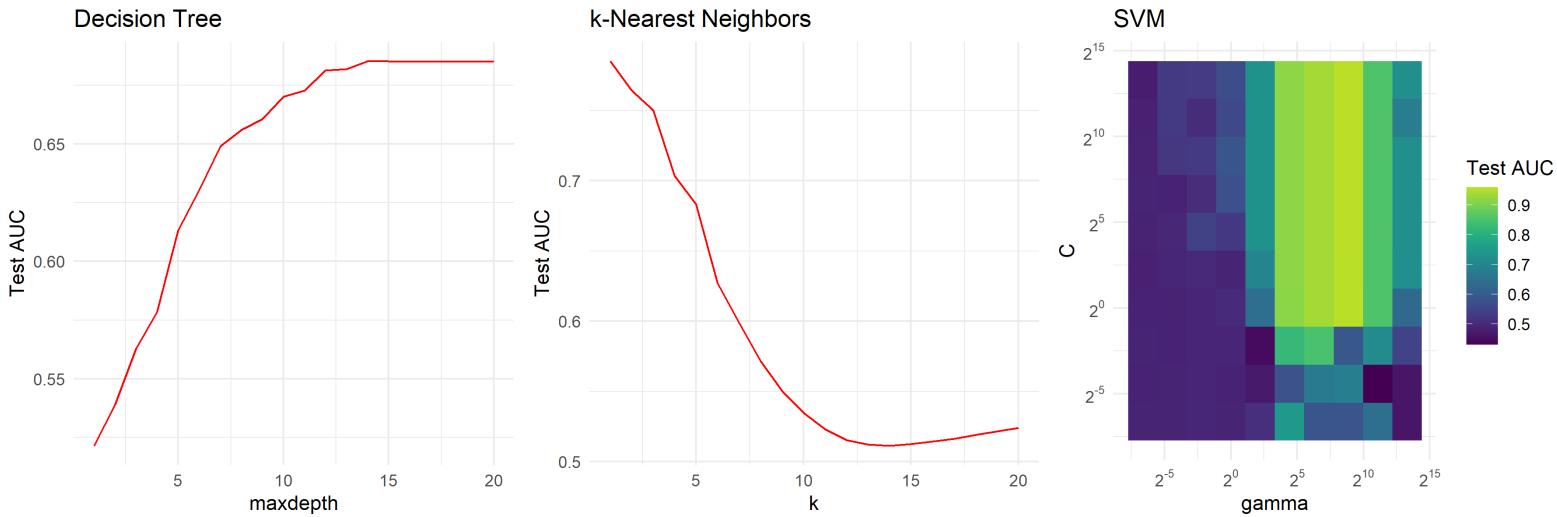
MOTIVATING EXAMPLE

- Given a data set, we want to train a classification tree.
- We feel that a maximum tree depth of 4 has worked out well for us previously, so we decide to set this hyperparameter to 4.
- The learner ("inducer") $\mathcal{I}$ takes the input data, internally performs **empirical risk minimization**, and returns a fitted tree model $\hat{f}(\mathbf{x}) = f(\mathbf{x}, \hat{\theta})$ of at most depth $\lambda = 4$ that minimizes empirical risk.



MOTIVATING EXAMPLE

- But many ML algorithms are sensitive w.r.t. a good setting of their hyperparameters, and generalization performance might be bad if we have chosen a suboptimal configuration.
- Consider a simulation example of 3 ML algorithms below, where we use the dataset *mlbench.spiral* and 10,000 testing points. As can be seen, variating hyperparameters can lead to big difference in model's generalization performance.



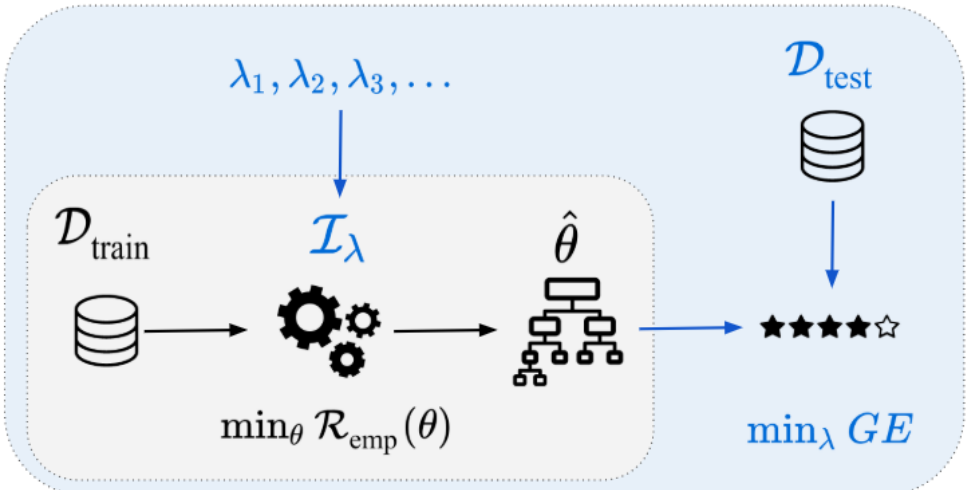
MOTIVATING EXAMPLE

For our example this could mean:

- Data too complex to be modeled by a tree of depth 4
- Data much simpler than we thought, a tree of depth 4 overfits

$\Longrightarrow$ Algorithmically try out different values for the tree depth. For each maximum depth λ, we have to train the model **to completion** and evaluate its performance on the test set.

- We choose the tree depth λ that is **optimal** w.r.t. the generalization error of the model.



MODEL PARAMETERS VS. HYPERPARAMETERS

It is critical to understand the difference between model parameters and hyperparameters.

Model parameters θ are optimized during training. They are an **output** of the training.

Examples:

- The splits and terminal node constants of a tree learner
- Coefficients θ of a linear model $f(\mathbf{x}) = \theta^\top \mathbf{x}$

MODEL PARAMETERS VS. HYPERPARAMETERS

In contrast, **hyperparameters** (HPs) λ are not optimized during training. They must be specified in advance, are an **input** of the training. Hyperparameters often control the complexity of a model, i.e., how flexible the model is. They can in principle influence any structural property of a model or computational part of the training process.

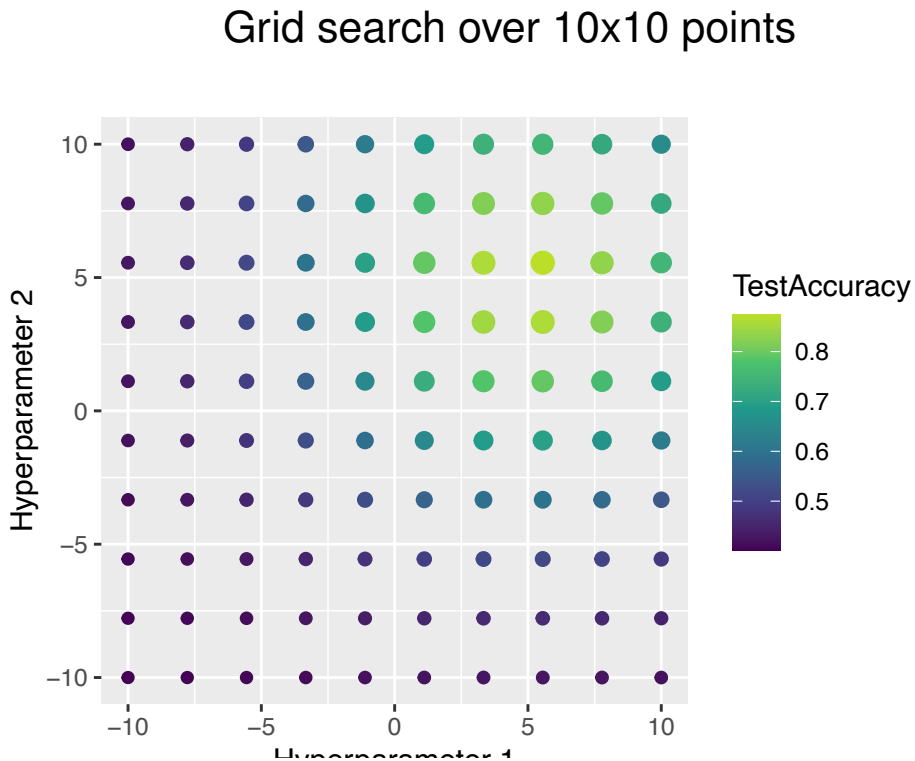
The process of finding the best hyperparameters is called **tuning**.

Examples:

- Maximum depth of a tree
- k and which distance measure to use for k-NN
- Number and maximal order of interactions to be included in a linear regression model
- Number of optimization steps if the empirical risk minimization is done via gradient descent

GRID SEARCH

- Simple technique which is still quite popular, tries all HP combinations on a multi-dimensional discretized grid
- For each hyperparameter a finite set of candidates is predefined
- Then, we simply search all possible combinations in arbitrary order



GRID SEARCH

Advantages

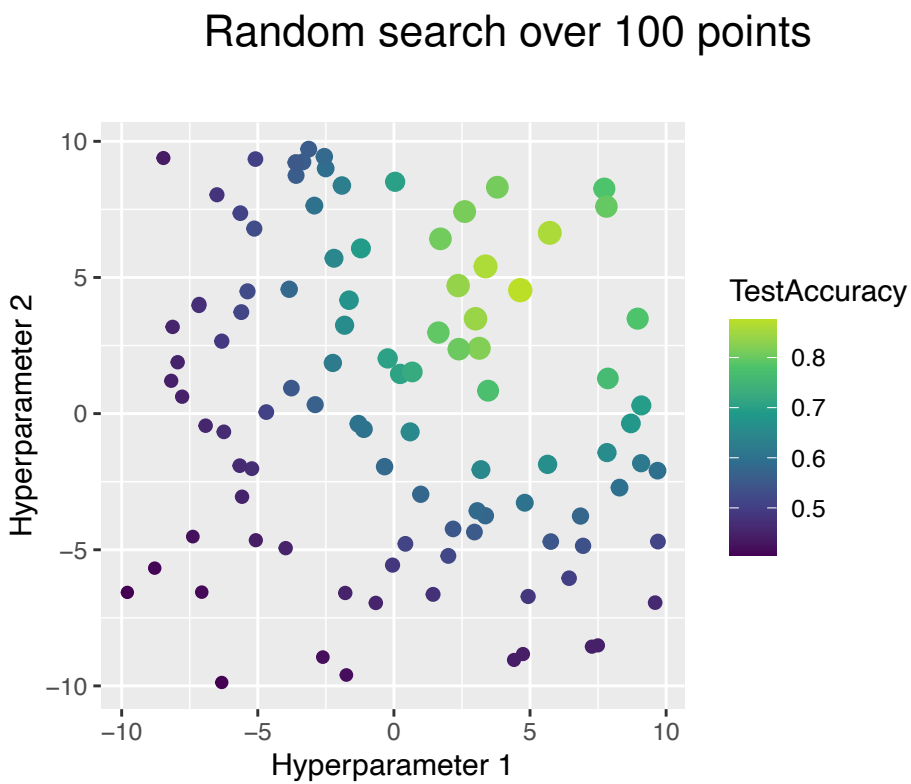
- Very easy to implement
- All parameter types possible
- Parallelizing computation is trivial

Disadvantages

- Scales badly: combinatorial explosion
- Inefficient: searches large irrelevant areas
- Arbitrary: which values / discretization?

RANDOM SEARCH

- Small variation of grid search
- Uniformly sample from the region-of-interest



RANDOM SEARCH

Advantages

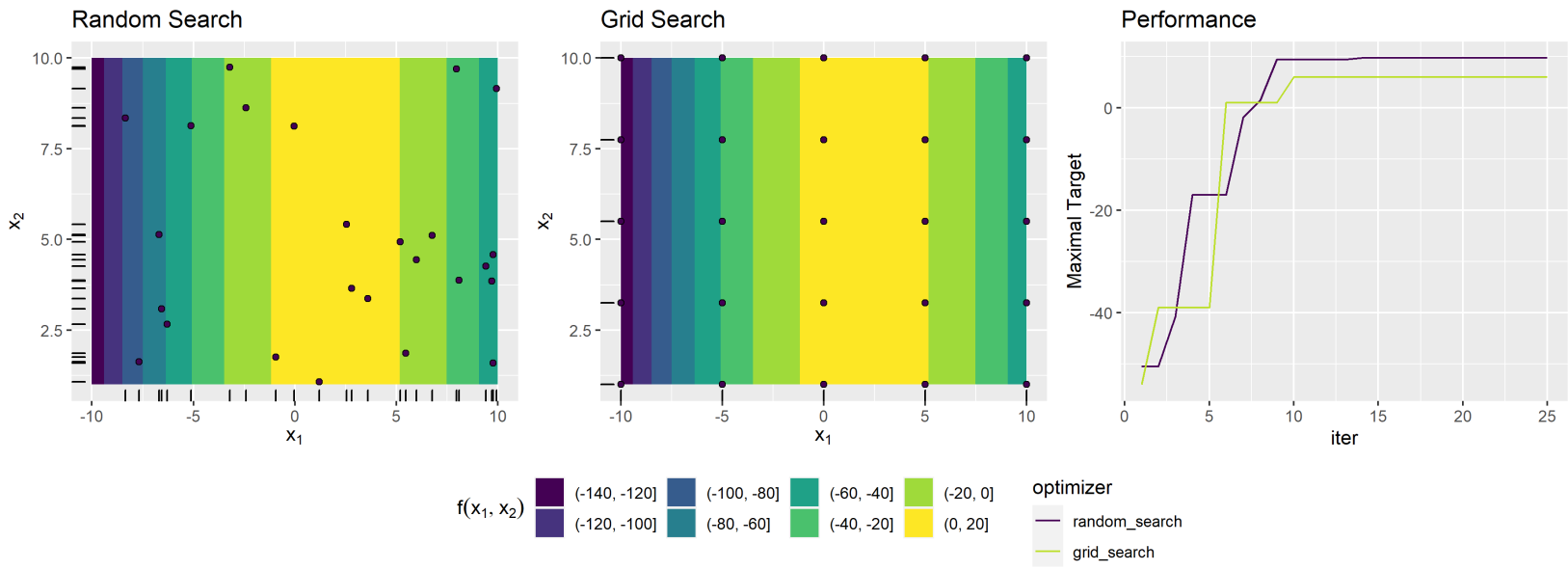
- Like grid search: very easy to implement, all parameter types possible, trivial parallelization
- Anytime algorithm: can stop the search whenever our budget for computation is exhausted, or continue until we reach our performance goal.
- No discretization: each individual parameter is tried with a different value every time

Disadvantages

- Inefficient: many evaluations in areas with low likelihood for improvement
- Scales badly: high-dimensional hyperparameter spaces need *lots* of samples to cover.

RANDOM SEARCH VS. GRID SEARCH

We consider a maximization problem on the function $f(x_1, x_2) = g(x_1) + h(x_2) \approx g(x_1)$, i.e. in order to maximize the target, x_1 should be the parameter to focus on.



$\Rightarrow$ In this setting, random search is more superior as we get a better coverage for the parameter x_1 in comparison with grid search, where we only discover 5 distinct values for x_1.

TUNING EXAMPLE

Tuning random forest with grid search/random search and 5CV on the `sonar` data set for AUC:

Hyperparameter	Type	Min	Max
`num.trees`	integer	3	500
`mtry`	integer	5	50
`min.node.size`	integer	10	100