# Regularization and optimisation for deep learning

**The Academy of AI 2018/19**

**Session 10**

Students for AI

# What is regularisation?

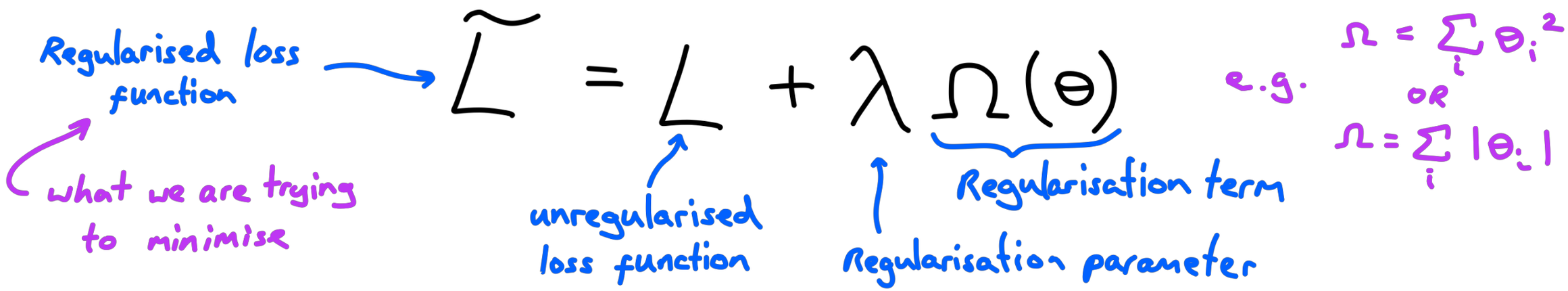
**Regularisation is any strategy designed to reduce the generalisation error of a model.**

How can we implement regularisation?

- Put constraints on weight values
- Add regularisation terms to the loss function
- Combine the predictions of several models

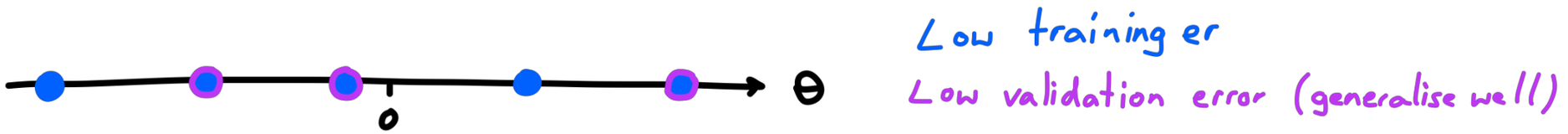
# Weight decay

Weight decay increases the loss function as the weights increase.




There are potentially lots of weight parameterisations that can result in low training error.

The range of values that weights can take is unbounded. Hence there are many more low training error parameterisations with weights far from zero compared to close to zero.




By randomly initialising our models weights, we may learn different low training error parameterisations each time. Some of these parameterisations will have low generalisation error, and some will not.

Weight decay regularisation puts a penalty on weights further from zero and hence puts a preference on learning parameterisations with weights in a certain region of weight space (closer to zero).

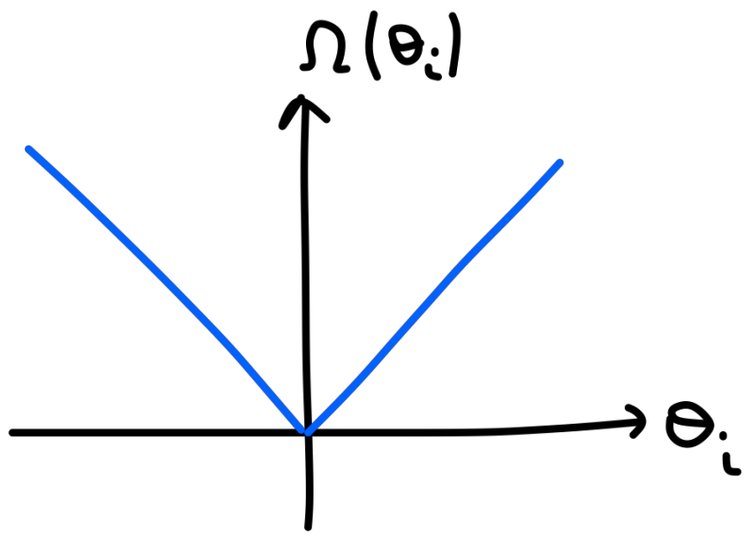
This decreases the parameterisation variance at the trade off of increasing its bias.

This increases the model's ability to generalise.

# Weight decay

$L_1$ norm

$$\Omega(\theta) = \sum_i |\theta_i|$$




$$\frac{\partial \Omega}{\partial \theta_i} = \begin{cases} 1, & \theta_i > 0 \\ -1, & \theta_i < 0 \end{cases}$$

Gradient magnitude is the same whether the parameter is close to zero or far from zero

This means that weights are consistently pushed towards zero

Parameters that weight unnecessary features will be pushed **all the way to zero.**

These features can then be discarded as inputs

L1 regularisation can hence be used as a **feature selector**

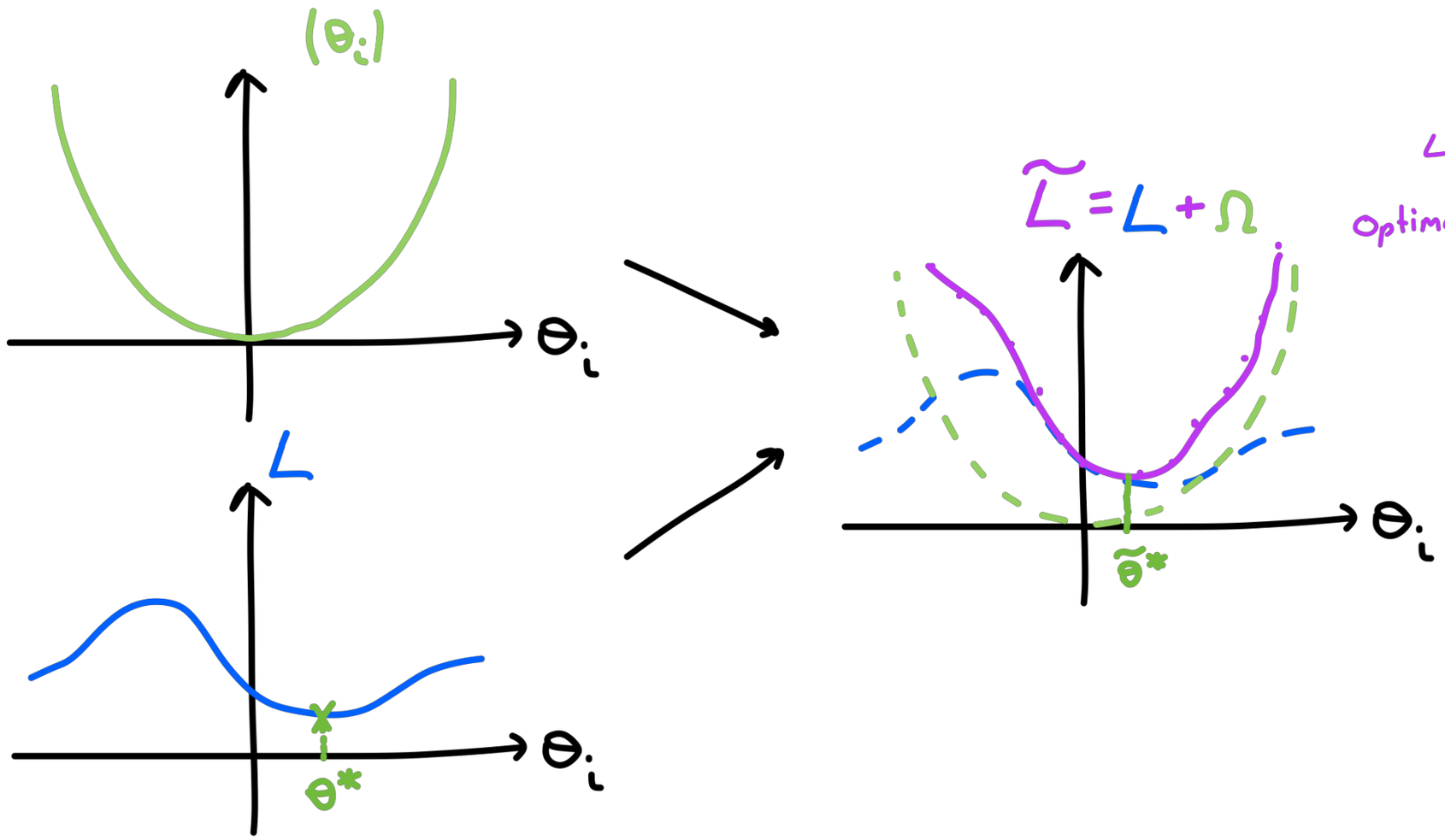
E.g contribution of age of previous owner of house would be pushed to zero when using it to predict house price, because it is irrelevant

# Weight decay

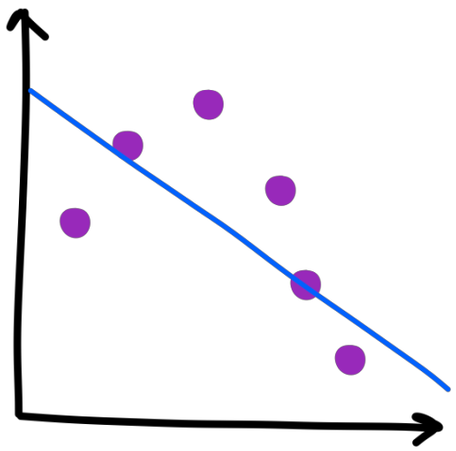
$L_p$ norm $= \sqrt[p]{\sum_i |\theta_i|^p}$

As weights get closer to zero, changing them affects the loss function less (gradient magnitude decreases nearer zero)

$L_2$ norm $\quad \Omega(\theta) = \sqrt{\sum_i |\theta_i|^2}$

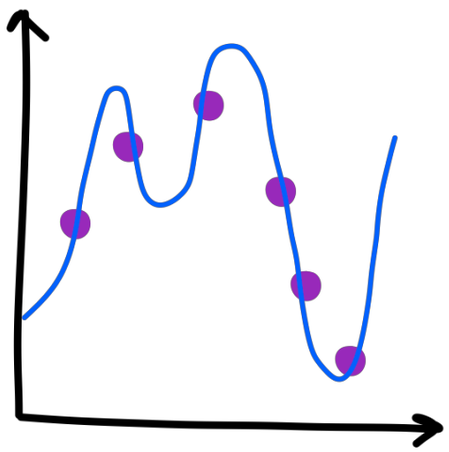


# Weight decay



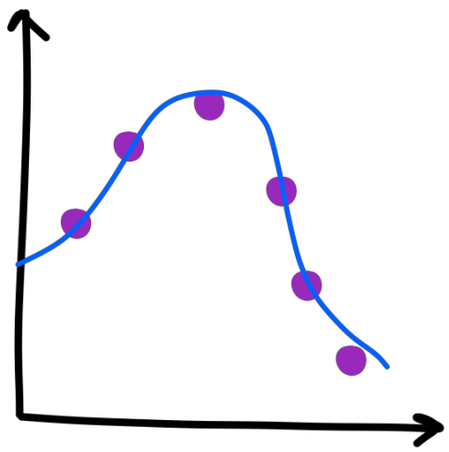
Underfit model with insufficient capacity

We need or models to have sufficient capacity to represent the input-output...



Overfit model with excessive capacity

...but they should not have characteristics of overly complex functions



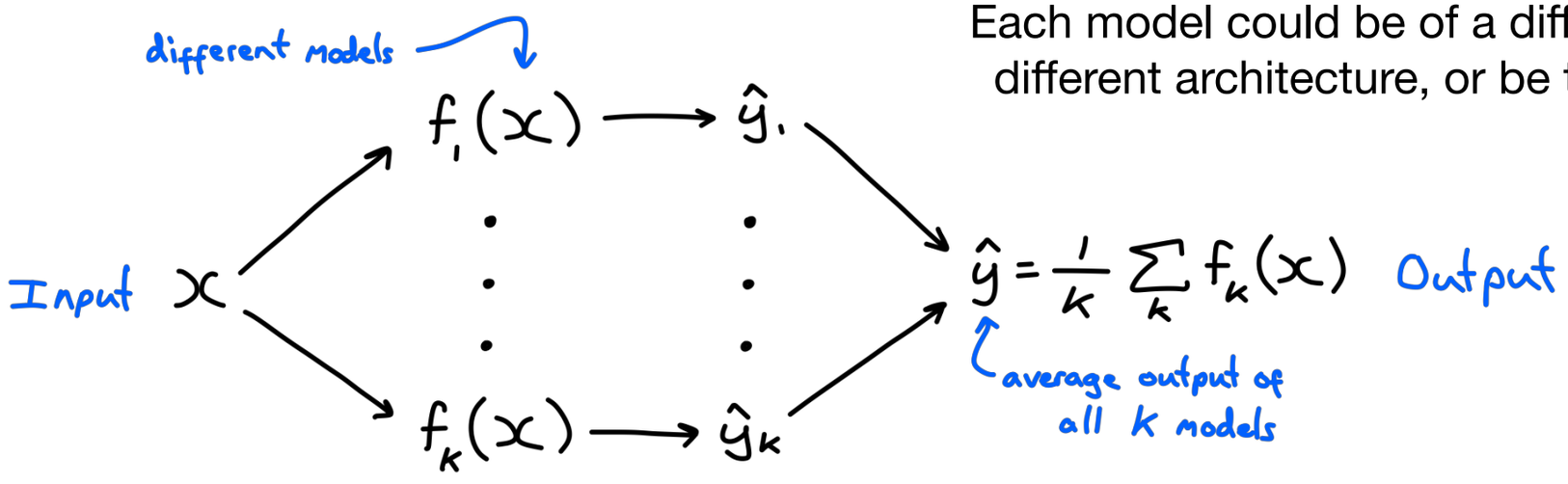
Regularised model with sufficient capacity

We can penalize overly complex models using regularisation

**Note: We don't want to regularise the biases (the y-intercept)**

# Ensembles

Ensemble methods average the output of more than one model to form a consensus




Each model could be of a different type, have a different architecture, or be trained differently

$\mathbb{E}[\varepsilon_i^2] = v$ ← expected square error of one model

$\mathbb{E}[\varepsilon_i \varepsilon_j] = c$ ← covariance of errors of models i & j

$$\mathbb{E}\left[\left(\frac{1}{k}\sum_i \varepsilon_i\right)^2\right] = \frac{1}{k^2}\mathbb{E}\left[\sum_i\left(\varepsilon_i^2 + \sum_{j\neq i}\varepsilon_i\varepsilon_j\right)\right] = \underbrace{\frac{1}{k}v + \frac{k-1}{k}c}$$

expected error decreases by factor of k when errors are uncorrelated

As long as the errors of each model are uncorrelated, the expected error will decrease

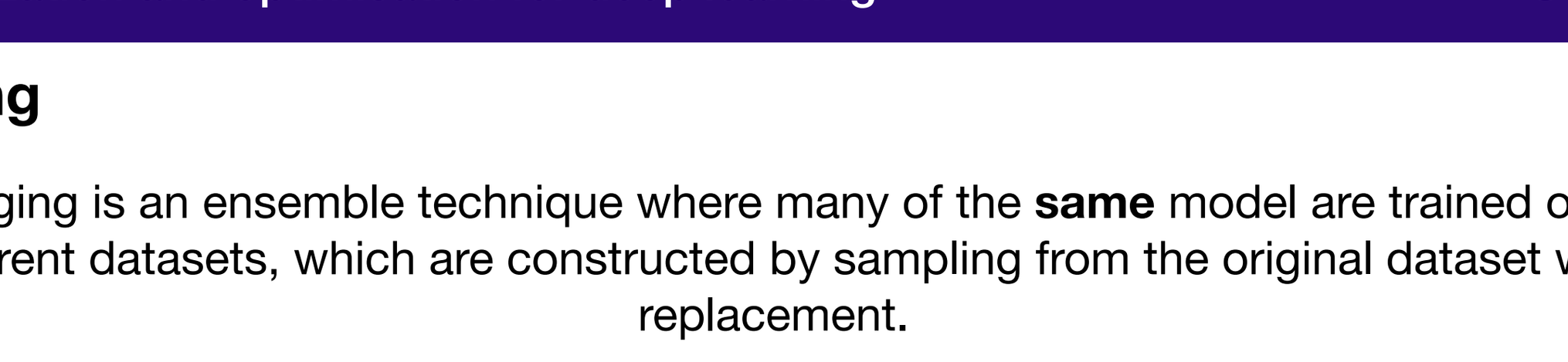
The crowd is smarter than the individual

# Bagging

Bagging is an ensemble technique where many of the **same** model are trained on K different datasets, which are constructed by sampling from the original dataset with replacement.

With high probability, some of the datasets will not contain certain examples and will contain duplicates.

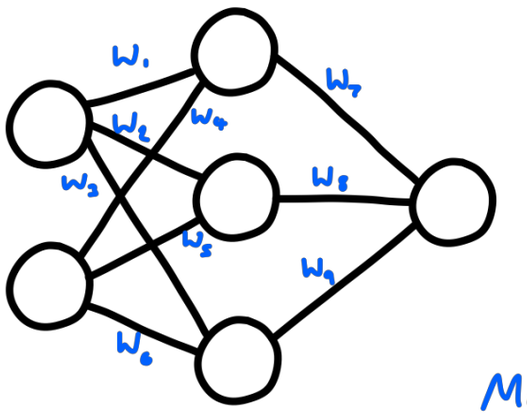
As such, the loss landscape will look different to each model and they may learn different features and depend on them to predict the right output.




These models depend on different features and could each be fooled by other unseen examples (a 6), but by bagging them together, they can combine what they've learnt and generalise.

# Dropout

Dropout randomly removes weights of neural networks (sets them to zero) during each forward pass of training

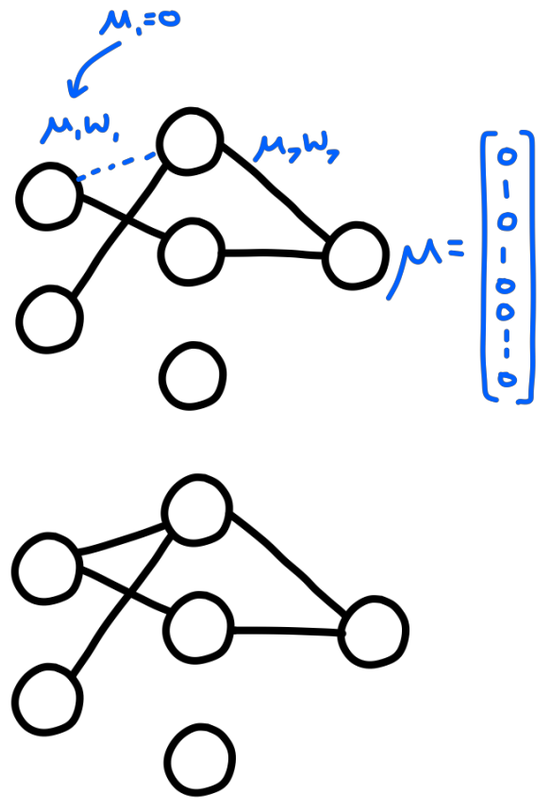


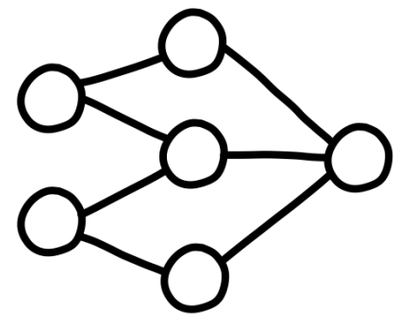

mask weights with probability p

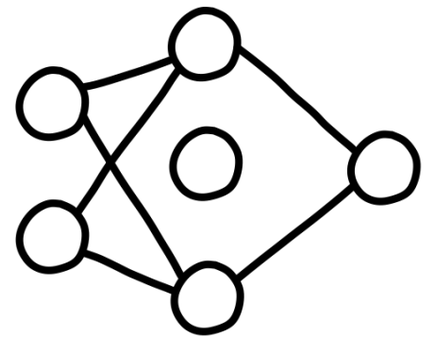
$$\text{Mask, } \mu = \begin{bmatrix} \mu_1 \\ \vdots \\ \mu_9 \end{bmatrix}$$

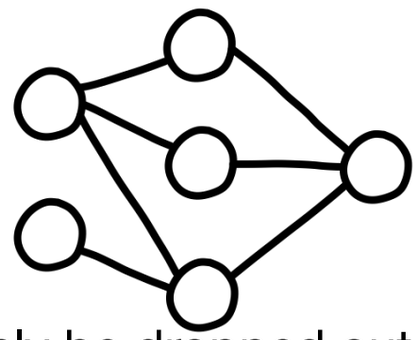
Without some weights, the model represents a different function

Subnetworks of original model
↳ generated by masking each weight







Dropout approximates bagging with an exponential number of different models

When training with dropout, the model cannot depend on any given feature as it may suddenly be dropped out.

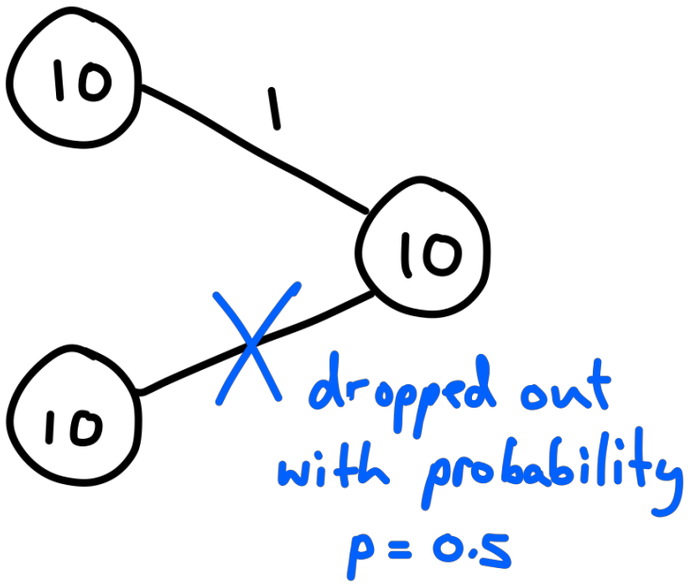
This makes it robust to learning features that do not depend on each other and also prevents the output depending on any particular feature

## Dropout

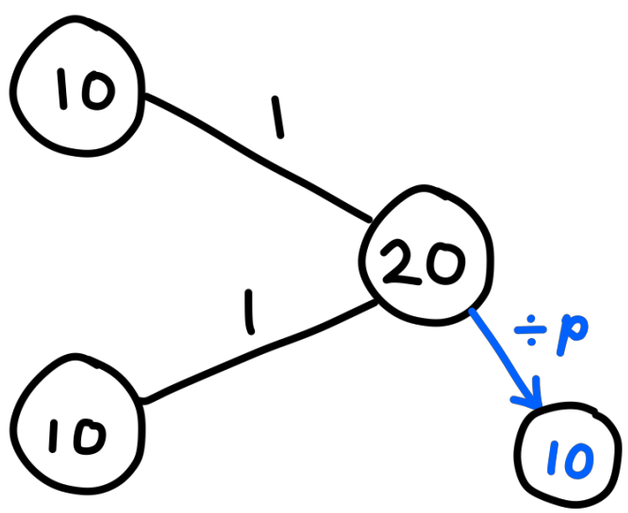
Once the model is trained, we want to use all of the learned weights for inference, and not drop any weights.

This means that models trained with dropout behave differently during training and deployment.

Training




Deployment



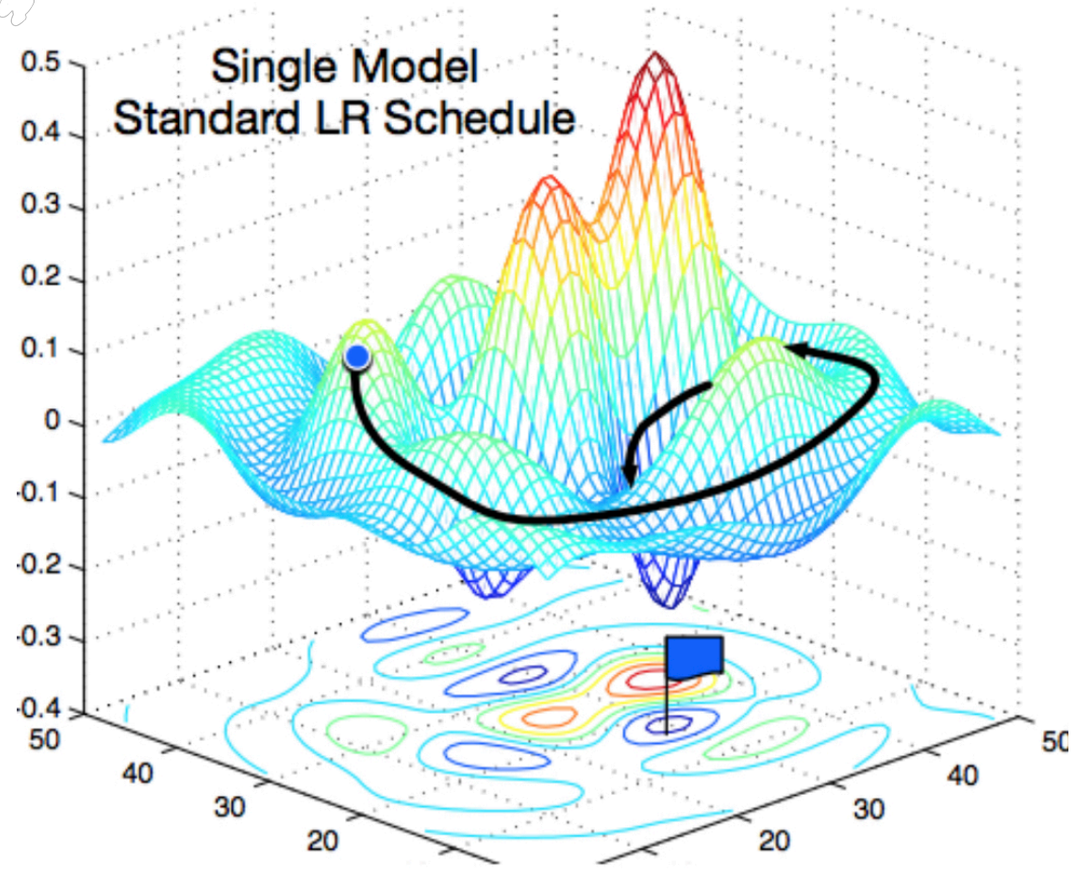

When deployed, we divide the input scores to nodes by the probability of a weight being dropped

This means that the biases and weights next applied to this node still portion a usefully learned transform

**Dropout provides an inexpensive way to approximate training and inference of exponentially many neural networks. This prevents co-adaptation of features and dependence on any specific features, which improves generalisation.**

# Gradient based learning

Loss




The optimisation techniques used for deep learning have the weights follow the downhill slope of the loss function to improve the model.

Simple gradient descent:

$$\theta \leftarrow \theta - \varepsilon \frac{\partial L}{\partial \theta}$$

learning rate

**The difference between learning and optimisation:**

Learning differs to pure optimisation because the true loss surface is not known - we don't know how our model will perform on every example that could exist because we don't have every example in our dataset. We only have a sample of examples, the distribution of which we hope is representative of all possible data points.

# Stochastic Gradient Descent (SGD) with momentum

Initialise weights, $\theta$ randomly & velocity, $V$ as zero

While stopping criteria not met:

sample minibatch of $m$ examples from training set

compute gradient $g \leftarrow \frac{1}{m} \nabla_\theta \sum_i L\left(f(x^{(i)}; \theta), y^{(i)}\right)$

compute velocity $v \leftarrow \alpha v - \varepsilon g$ ← velocity increases in downhill direction

"weight" maintains current vel → momentum parameter; learning rate

update weights $\theta \leftarrow \theta + v$ ← weights move in direction of velocity

# AdaGrad

$r = 0$ ← gradient accumulation variable vector — this is the core of AdaGrad

while stopping criteria not met:

sample minibatch

compute gradient $g = \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$

accumulate squared gradient $r \leftarrow r + \overbrace{g \odot g}^{g^2}$ ← $r = \begin{bmatrix} r_1 + \left(\frac{\partial L}{\partial \theta_1}\right)^2 \\ \vdots \\ r_n + \left(\frac{\partial L}{\partial \theta_n}\right)^2 \end{bmatrix}$ r = accumulated squared gradient

update $\theta$ by: $\theta \leftarrow \theta - \frac{\varepsilon}{\delta + \sqrt{r}} \odot g$ → $\Delta\theta$

maybe $10^{-7}$ ⟶ small constant for numerical stability

↳ normally $-\varepsilon g$

↳ so scaled by $\frac{1}{\delta + \sqrt{r}}$

Intuition: each model parameter in $\theta$ has its own

learning rate = $\frac{\varepsilon}{\delta + \sqrt{r}}$ ← these learning rates are adaptive because $r$ changes

gradient accumulator will be lower for more flat areas, so learning rate remains high & progress is made, even though gradient here is small.

Learning rate $\varepsilon$ can be initialised to a greater value; Divergence will probably occur along directions where the gradient is high initially – but the learning rate will be adapted in this direction, by its division by $\delta + \sqrt{r}$

← large here ⟶ $\varepsilon$ decreased ⟶ convergence

The problem with AdaGrad is that the learning rate cannot be recovered once large partial derivatives are encountered.

Hence learning rates can be made too small before reaching a convex region of the loss landscape suitable for convergence

# RMSProp

RMSProp works like AdaGrad, but uses an exponentially decaying average to discard gradients from the extreme past so that it can converge quickly after finding a convex bowl.

More recent gradient observations contribute more to the gradient accumulation

$r = \vec{0}$ ← gradient accumulation vector variable

while stopping criterion not met:

sample minibatch

compute gradient $g = \frac{1}{M} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$

accumulate squared gradient $r \leftarrow \rho r + (1-\rho)\, g \odot g$

update $\theta$ by: $\theta \leftarrow \theta - \frac{\varepsilon}{\sqrt{\delta + r}} \odot g$

core of RMSProp

$\rho$ = decay rate

exponentially decaying average

$r \leftarrow \rho(r - g \odot g) + g \odot g$

# Adam (adaptive moments)

first moment

$s = 0$

$r = 0$ } initialise moment variables

2nd moment

$t = 0$ initialise time step counter

while stopping criteria not met:

sample minibatch

compute gradient $g \leftarrow \frac{1}{m} \nabla_\theta \sum_{i=1}^{m} L\left(f(x^{(i)}; \theta), y^{(i)}\right)$

$t \mathrel{+}= 1$ update time

Update biased first moment estimate $s \leftarrow \rho_1 s + (1-\rho_1) g$ ← like velocity

Update biased second moment estimate $r \leftarrow \rho_2 r + (1-\rho_2) g \odot g$ ← learning rate decay

Correct first moment's bias $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$

Correct second moment's bias $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$ } unbiased estimates

↳ corrected, unlike in RMSProp

Compute update $\theta \leftarrow \theta - \varepsilon \frac{\hat{s}}{\sqrt{\hat{r}} + \delta}$ ← provides stability

# Batch normalisation

When we calculate rates of change of loss with respect to weights, we assume that all other weights remain constant.

In practice, we update all weights simultaneously.

This means that we move to a parameterisation where weight updates may have been uncoordinated

Batch normalisation coordinates the weight updates, and makes training much more stable for deep models.

Similarly to how we usually normalise our features at the input of the model, batch normalisation normalises the activations of each hidden unit, for each batch's forward pass.

$$H' = \frac{H - \mu}{\sigma}$$

During deployment, we normalise each layers activation by a moving average of the computed means and standard deviations of the training batches

## Our goal

**We want to create hackers, who can use AI to solve real problems**

**So please, let us know what you want**

**What do you want?!?**