Deep Learning

Lecture 2

Recap

- Multi-layer perceptron
  - Motivation
  - Activation function
  - Properties
- Gradient calculation
  - Numerical calculation
  - Automatic differentiation
  - Manual differentiation

Contents

How to use a gradient for optimization?

- Classical gradient descent
- Modified gradient descent
- Regularization techniques

Neural network optimization

Optimization problem

- Typical objectives in machine learning are an average over training cases of case-specific losses:

$$F(x) = \frac{1}{n}\sum_{i=1}^{n} f_i(x) \to \min_x, n \gg 1, x \in \mathbb{R}^q$$

- We need some optimization method. What about gradient descent?

$$x_{k+1} = x_k - \alpha_k \nabla F(x_k)$$

- Training dataset size can be **very** big, and so computing the gradient gets expensive:

Function	Calculation cost
$f_i(x)$	$\mathcal{O}(q)$
$\nabla f_i(x)$	$\mathcal{O}(q)$
$F(x)$	$\mathcal{O}(nq)$
$\nabla F(x)$	$\mathcal{O}(nq)$

Stochastic Gradient Descent (SGD)

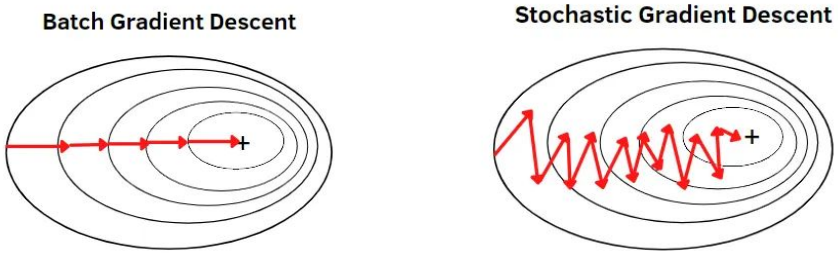
Solution: what if you use not all samples, but just one?

$$i_k \sim \mathcal{U}(1, \dots, n)$$
$$g_k = \nabla f_{i_k}(x_k)$$
$$x_{k+1} = x_k - \alpha_k g_k$$

Expected value of gradient is equal to the true gradient!

$$\mathbb{E} g_k = \nabla F(x_k)$$

Problem: high variance of gradient



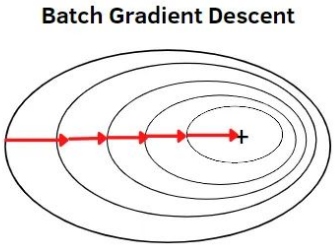
Mini-Batch Gradient Descent (Batch SGD)

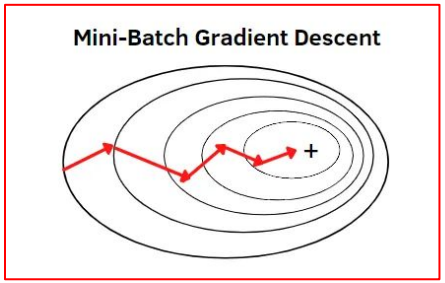
Idea: randomly subsample a “mini-batch” of training cases, and estimate gradient as:

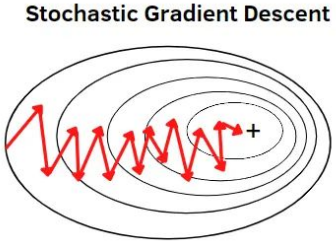
$$I_k \subset \mathcal{U}(1, \ldots, n)$$

$$g_k = \frac{1}{|I_k|} \sum_{i \in I_k} \nabla f_{i_k}(x_k)$$

$$x_{k+1} = x_k - \alpha_k g_k$$







Expected value of gradient is equal to the true gradient!

$$\mathbb{E} g_k = \nabla F(x_k)$$

But the variance of the gradient is reduced!

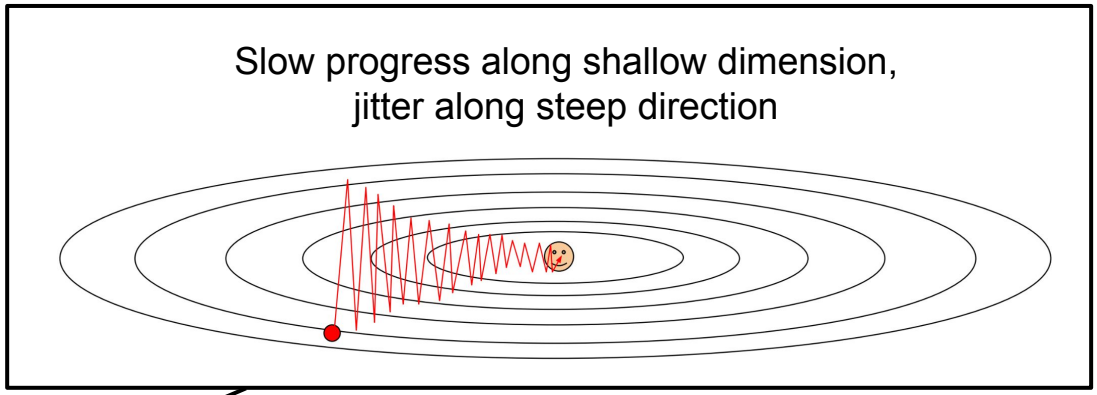
SGD: pros and cons

Advantages:

- Computational Efficiency
- Memory Efficiency
- Avoidance of Local Minima
- Frequent Updates

Disadvantages:

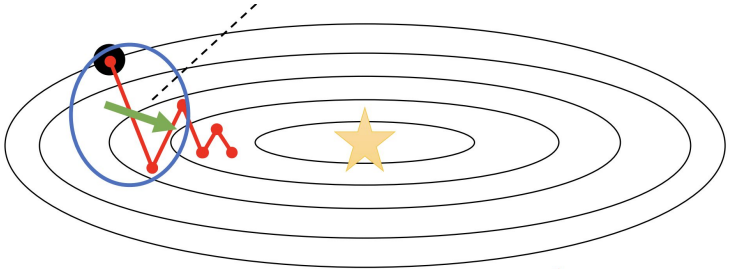
- Noisy Updates
- More Iterations Required
- Different progress for various directions



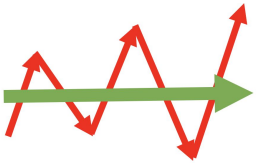
How can we deal with it?

Momentum

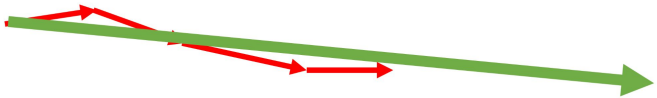
- Averaging together successive gradients seems to yield a much better direction!



- **Intuition:**
  - If successive gradient steps point in **different** directions, we should **cancel off** the directions that disagree



  - If successive gradient steps point in **similar** directions, we should **go faster** in that direction



SGD + Momentum update

SGD:

$$x_{k+1} = x_k - \alpha_k g_k$$

SGD + Momentum:

$$y_{k+1} = \gamma y_k + g_k$$

$$x_{k+1} = x_k - \alpha_k y_{k+1}$$

- Build up “velocity” as a running mean of gradients
- Effect of the gradient is to increment the previous velocity
- The weight change is equal to the current velocity

Can be seen as an exponential moving average (EMA):

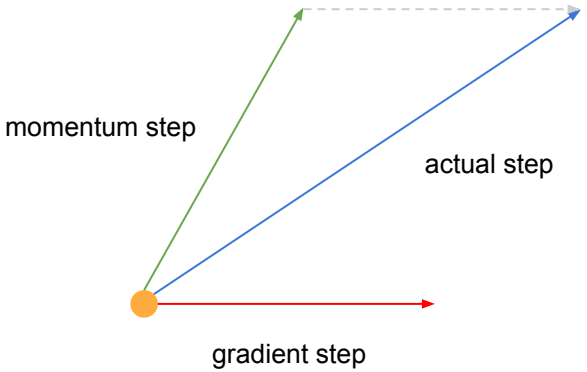
$$y_{k+1} = \gamma y_k + g_k = \ldots =$$

$$= \gamma^k g_1 + \ldots + \gamma g_{k-1} + g_k$$

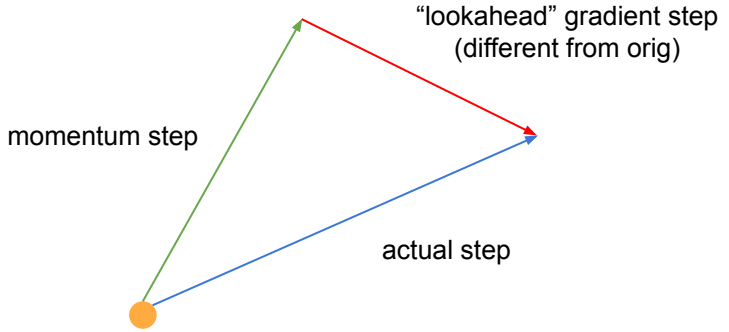
Nesterov momentum

Compute gradient where you **will be**, not where you **are**

Momentum update



Nesterov momentum update

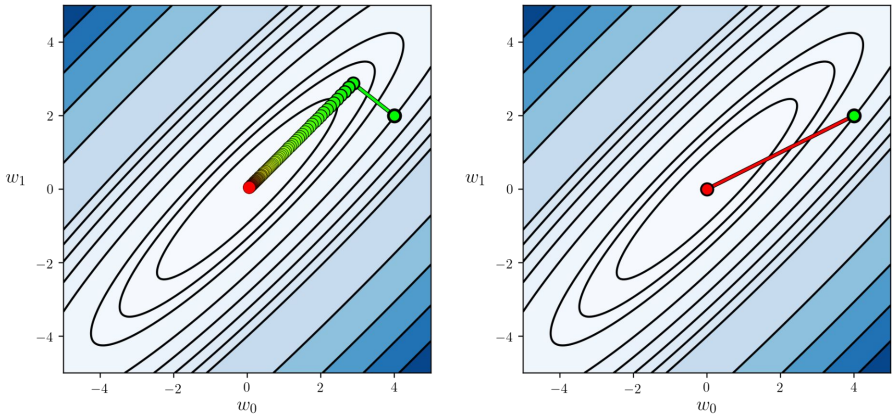


How to choose a direction?

- The best method for quadratic forms is a **Newton method**
- Use the second-order information
- Adaptive coefficients for the gradient components

Newton method:

$$x_{k+1} = x_k - \alpha_k (\nabla^2 f(x_k))^{-1} \nabla f(x_k)$$



AdaGrad

Idea: “normalize” out the magnitude of the gradient along each dimension

- Save the squared gradient sums

$$x_{k+1,i} = x_{k,i} - \frac{\alpha_k}{\sqrt{v_{k,i} + \varepsilon}} g_{k,i}$$

$$v_{k,i} = \sum_{j=0}^{k} g_{j,i}^2$$

- Learning rate effectively “decreases” over time
- But this only works if we find the optimum quickly

RMSProp

- We can use momentum to prevent convergence from slowing down
- Use a running mean instead of saving all previous gradients

AdaGrad

$$x_{k+1,i} = x_{k,i} - \frac{\alpha_k}{\sqrt{v_{k,i} + \varepsilon}} g_{k,i}$$

$$v_{k,i} = \sum_{j=0}^{k} g_{j,i}^2$$

RMSProp

$$x_{k+1,i} = x_{k,i} - \frac{\alpha_k}{\sqrt{v_{k,i} + \varepsilon}} g_{k,i}$$

$$v_{k,i} = \beta v_{k-1,i} + (1 - \beta) g_{k,i}^2$$

EMA

Adam

ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION

Diederik P. Kingma*
University of Amsterdam, OpenAI
dpkingma@openai.com

Jimmy Lei Ba*
University of Toronto
jimmy@psi.utoronto.ca

Citations: 189,279

- How to make the perfect algorithm?
- Combine the best ideas: Adam = RMSProp + Momentum

$$x_{k+1,i} = x_{k,i} - \frac{\alpha_k}{\sqrt{v_{k,i} + \varepsilon}} \mu_{k,i}$$

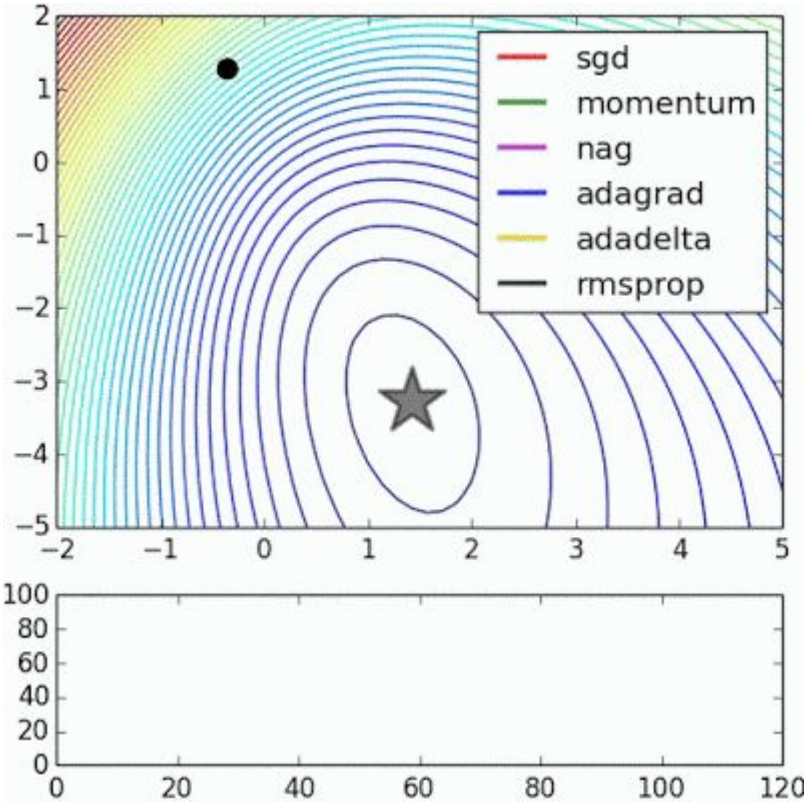
$$v_{k,i} = \beta_1 v_{k-1,i} + (1 - \beta_1) g_{k,i}^2$$ ← EMA adaptive learning rate

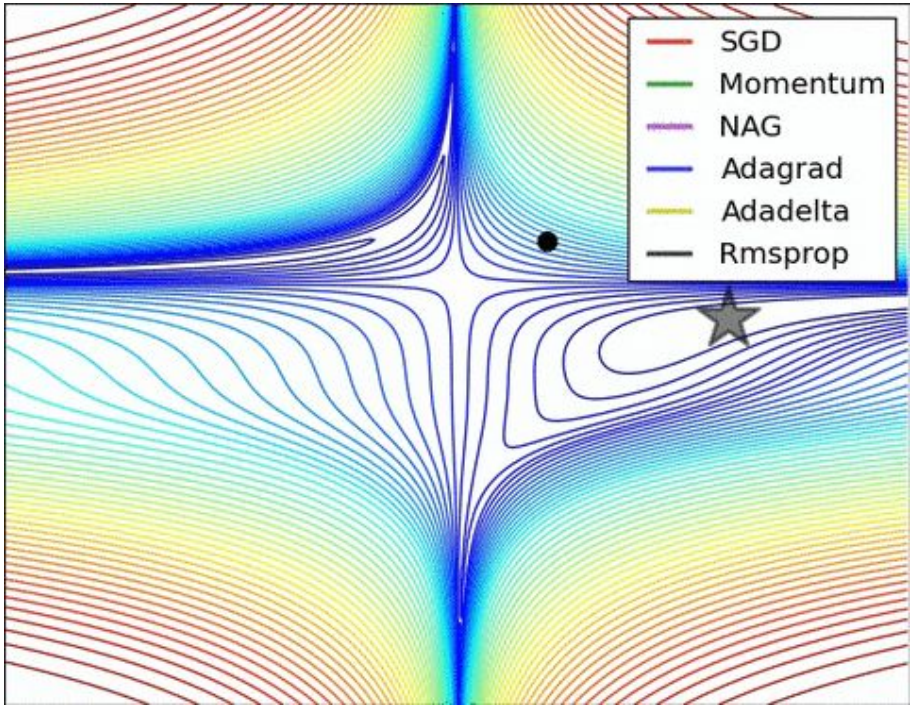
$$\mu_{k,i} = \beta_2 \mu_{k-1,i} + (1 - \beta_2) g_{k,i}$$ ← Momentum

Practical recommendations:

$$\alpha = 0.001$$
$$\beta_1 = 0.9$$
$$\beta_2 = 0.999$$

Comparison





Source: [A journey into Optimization algorithms for Deep Neural Networks]

Overall about optimizers

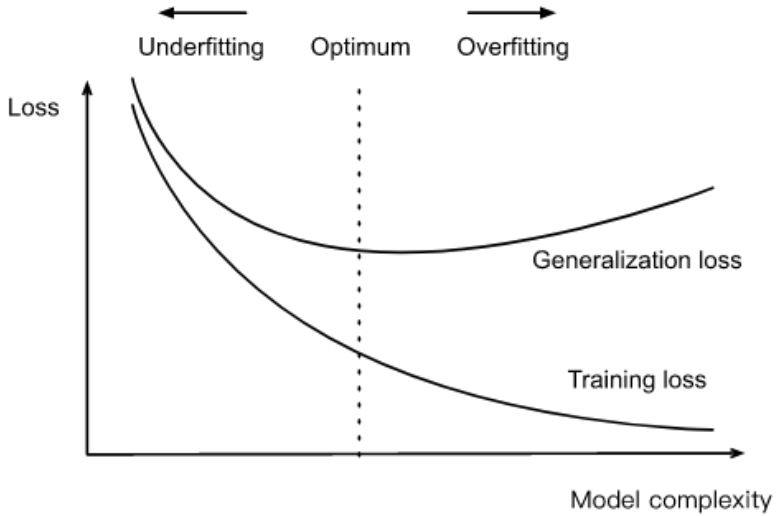
Each of the proposed optimization methods has its own advantages and disadvantages.

- Methods with **momentum** tend to converge to a solution more gradually
- May "overshoot" the desired value
- May oscillate around saddle points before reaching the optimal path

- Methods with **adaptive learning rates** converge faster and are more stable, reducing random fluctuations
- Moreover, algorithms without adaptive learning rates may be more challenging to escape from local minima

Regularization

Model regularization



From machine learning, we know that model complexity affects generalization ability of the model. If the model is too complex, (neural networks). So, it has to be regularized to achieve better generalization

About double descent https://mlu-explain.github.io/double-descent
Grokking: link

Weight decay

The first technique for model regularization which comes to mind is L2 Loss regularization. In deep learning, it's called **weight decay.**

$$L(\mathbf{w}, b) + \frac{\lambda}{2}\|\mathbf{w}\|^2,$$

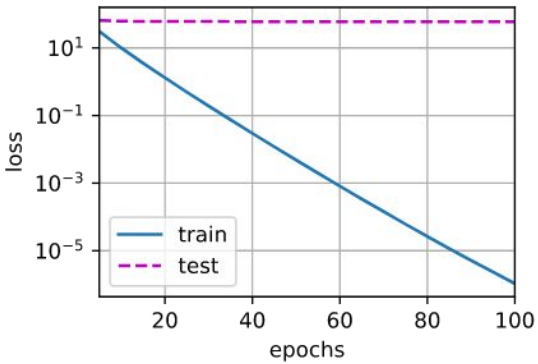
If we write out the equation for weight updates, we can directly understand why it's weight decay.

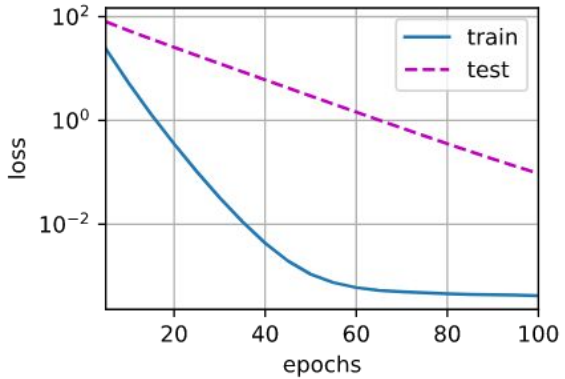
$$\mathbf{w} \leftarrow \underbrace{(1 - \eta\lambda)\,\mathbf{w}}_{\text{weight decay}} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right).$$

Example

Consider the simple example

$$y = 0.05 + \sum_{i=1}^{d} 0.01 x_i + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.01^2).$$





Weight decay w/ adaptive optimizers

Weight decay w/ SGD

$$\theta_{t+1} = \theta_t - \alpha \nabla f^{reg}(\theta_t) = (1 - \alpha\lambda)\theta_t - \alpha \nabla f(\theta_t)$$

Weight decay w/ Adam

$$\theta_{t+1} = \theta_t - \alpha \frac{\beta_1 m_{t-1} + (1 - \beta_1) g_t}{\sqrt{\beta_2 v_{t-1} + (1 - \beta_2) g_t^2} + \varepsilon}$$

where $g_t = \nabla f_t(\theta_{t-1}) + \lambda \theta_{t-1}$

There is no weight decay effect!

https://www.fast.ai/posts/2018-07-02-adam-weight-decay.html#adamw
https://github.com/GitYCC/machine-learning-papers-summary/blob/master/optimization-training-techniques/AdamW.md

Decoupled weight decay (AdamW)

Solution: add weight decay regularization manually

Algorithm 2 Adam with L_2 regularization and Adam with decoupled weight decay (AdamW)

given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$
initialize time step $t \leftarrow 0$, parameter vector $\boldsymbol{\theta}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
repeat
  $t \leftarrow t + 1$
  $\nabla f_t(\boldsymbol{\theta}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{\theta}_{t-1})$ ▷ select batch and return the corresponding gradient
  $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{\theta}_{t-1}) + \lambda \boldsymbol{\theta}_{t-1}$
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1) \boldsymbol{g}_t$ ▷ here and below all operations are element-wise
  $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2) \boldsymbol{g}_t^2$
  $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \beta_1^t)$ ▷ β_1 is taken to the power of t
  $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \beta_2^t)$ ▷ β_2 is taken to the power of t
  $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, or also be used for warm restarts
  $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \eta_t \left(\alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) + \lambda \boldsymbol{\theta}_{t-1} \right)$
until *stopping criterion is met*
return optimized parameters $\boldsymbol{\theta}_t$

Dropout

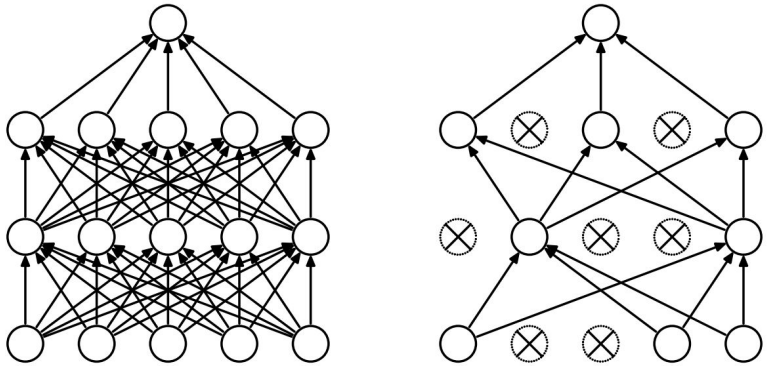
Let's consider the following procedure:

The feed-forward operation of a standard neural network

$$
\begin{aligned}
z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \mathbf{y}^l + b_i^{(l+1)}, \\
y_i^{(l+1)} &= f(z_i^{(l+1)}),
\end{aligned}
$$

With dropout, the feed-forward operation becomes

$$
\begin{aligned}
r_j^{(l)} &\sim \text{Bernoulli}(p), \\
\tilde{\mathbf{y}}^{(l)} &= \mathbf{r}^{(l)} * \mathbf{y}^{(l)}, \\
z_i^{(l+1)} &= \mathbf{w}_i^{(l+1)} \tilde{\mathbf{y}}^l + b_i^{(l+1)}, \\
y_i^{(l+1)} &= f(z_i^{(l+1)}).
\end{aligned}
$$



(a) Standard Neural Net (b) After applying dropout.

How it works visually

So, in average, we have the following hidden state

$$\mathbb{E}\tilde{\mathbf{y}}^{(l)} = (1-p)y^{(l)}$$

How it should work on the inference?

Why does it work?

1. It learns a large ensemble of models. By doing dropout, we implicitly create a huge number of models, and from ML we know that ensembles are better than single model

2. Randomly selecting different neurons ensure that neurons are unable to learn the co-adaptations and prevent overfitting.

Recap

- Gradient descent for neural networks
  - Stochasticity
  - Momentum
  - Adaptive methods
- Weight decay
- Dropout

TIME FOR A BREAK