Fitting Neural Networks (Chap11 in [HO])

- Main challenges of training deep neural networks (5+ hidden layers)
  - "Vanishing gradients" & "exploding gradients problem"
  - With such a large network, the training will be extremely slow
  - A model with this many parameters (1m+) has a high risk of overfitting

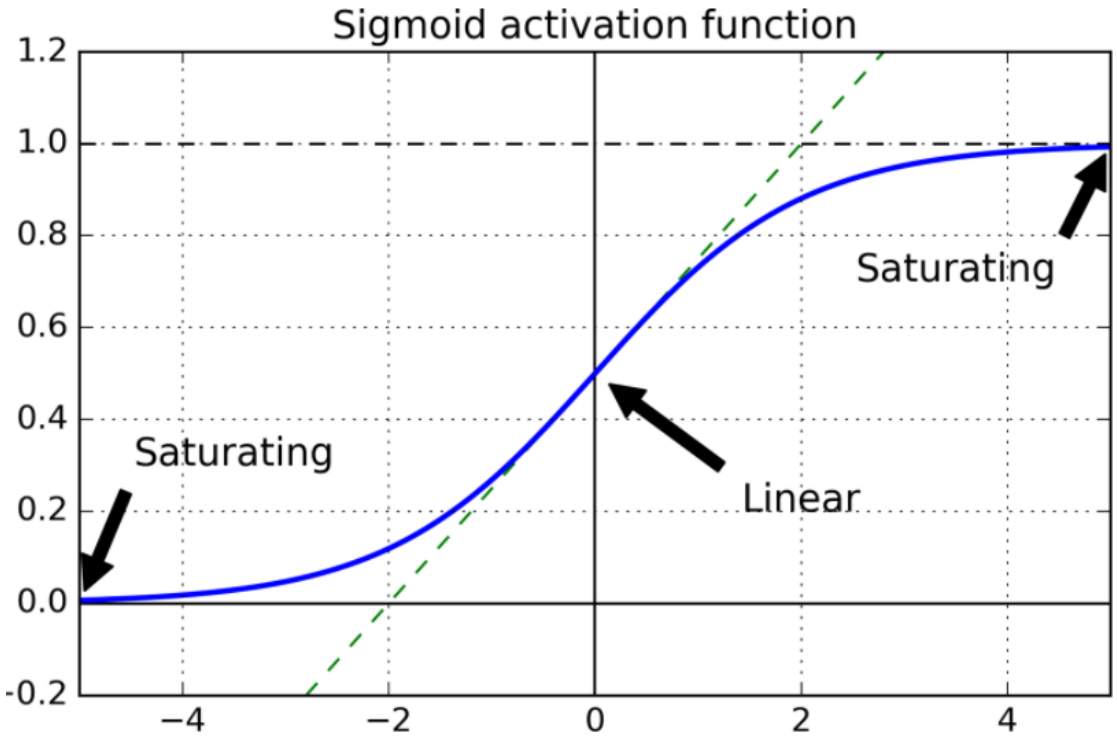
Vanishing and exploding gradients

- Vanishing gradients problem
  - Backpropagation algorithm works by going from the output layer to the input layer, propagating the error gradient on the way
  - Unfortunately, gradients often get smaller and smaller as the algorithm progresses down to the lower layers
    - As a result, the Gradient Descent update leaves the lower layer connection weights virtually unchanged, and training never converges to a good solution - Vanishing Gradients Problem
- Exploding gradients problem
  - The opposite of the Vanishing Gradients Problem - the gradients can grow bigger and bigger and then the algorithm diverges

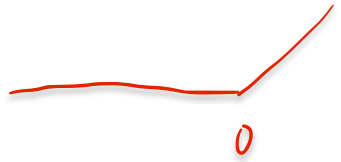
- Possible Solutions
  - First one, was only proposed in 2010 (by Xavier Glorot):
  - He found that the vanishing gradient problem can be dealt with by making sure that the variance of the output of each layer does not exceed the variance of its inputs by:
    - replacing the normal distribution (mean 0, deviation 1) random initialization of the weights with

      Xavier initialization:
      normal distribution with mean 0 and a custom deviation to try to equal the variance of the inputs and outputs
    - replacing the logistic sigmoid activation function (diverging from biological neurons)

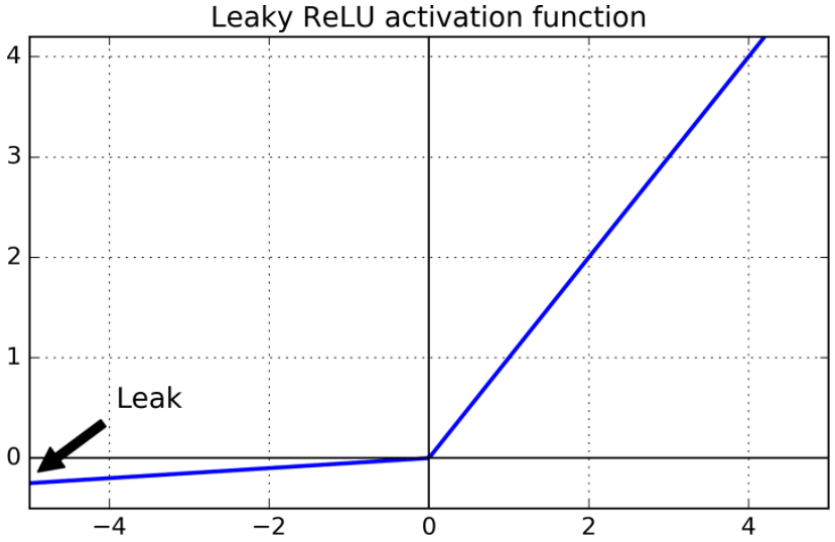
- Why replace the sigmoid function which was initially used for activation? The logistic sigmoid activation function saturates

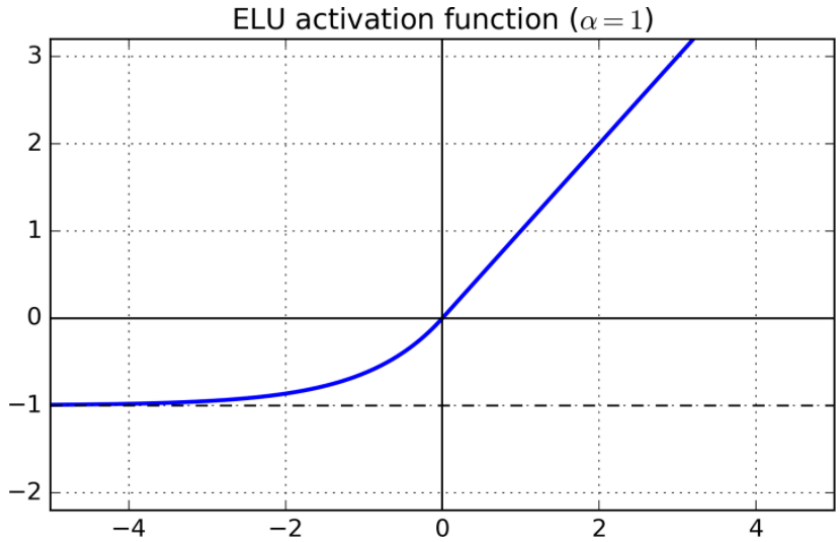


- There are a few good candidates
  - ReLU – but that has the dying neurons issue



  - Leaky ReLU
  - ELU (exponential linear unit)





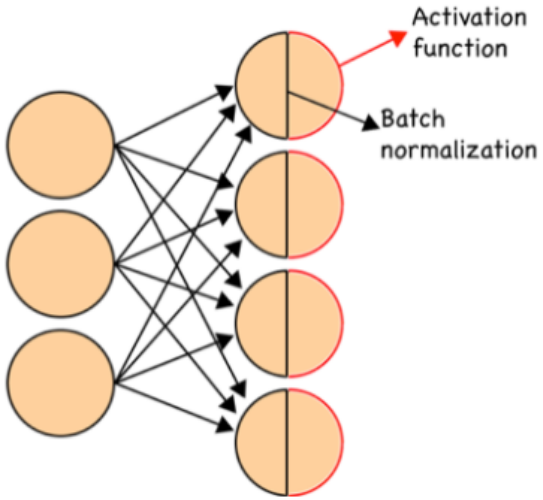
- Recommended preference for activation (if no other reasons are present)

  ELU > leaky ReLU > ReLU > Tanh > Logistic

- Xavier and He initialization and the Leaky ReLU / ELU activation can decrease the vanishing / exploding gradients problem at the beginning of the training, it doesn't guarantee that it won't come back during training
- Batch Normalization (2015):

- adding a new operation to the model just before the activation function of each layer to zero-center and normalize the inputs, then scale and shift the results using two new parameters per layer (~scaling + shifting)

- increases the training speed by achieving accuracy faster
- acts as a regularizer
- but the training steps are slower and inference is slower too



$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$$

Equation 11-3. Batch Normalization algorithm

1. $\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$

2. $\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} \left(\mathbf{x}^{(i)} - \mu_B\right)^2$

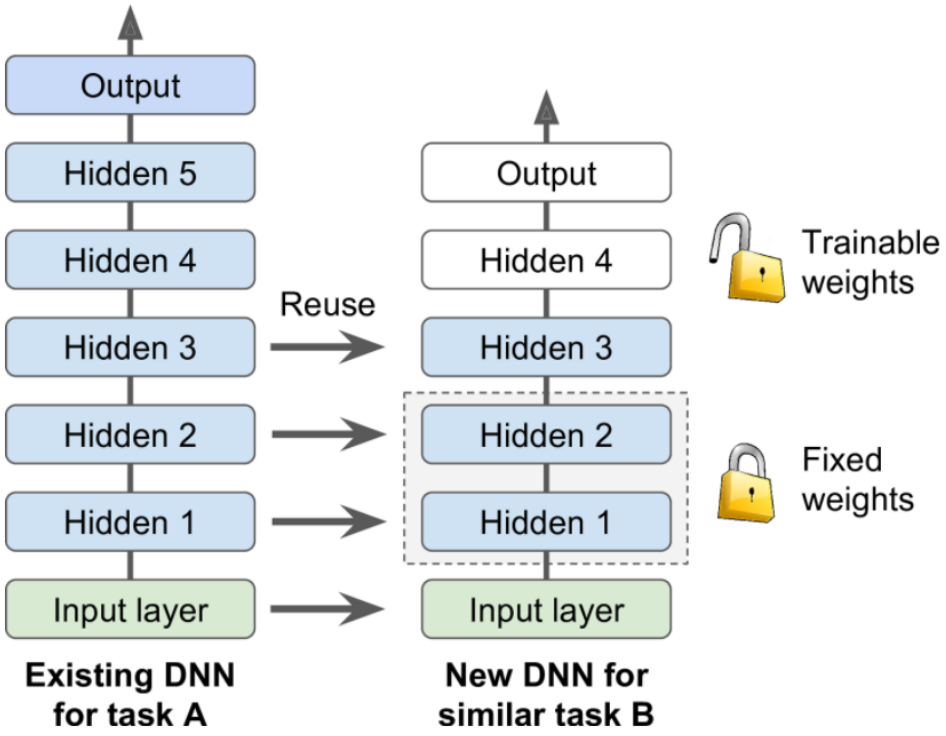
3. $\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}$

4. $\mathbf{z}^{(i)} = \gamma \otimes \hat{\mathbf{x}}^{(i)} + \beta$

- μ_B is the vector of input means, evaluated over the whole mini-batch *B* (it contains one mean per input).
- σ_B is the vector of input standard deviations, also evaluated over the whole mini- batch (1 st. dev. per input).
- m_B is the number of instances in the mini-batch.
- the $\hat{x}^{(i)}$ vector of zero-centered and normalized inputs for instance i.
- γ is the output scale parameter vector for the layer (it contains one scale parameter per input)
- $\otimes$ represents element-wise multiplication
- β is the output shift (offset) parameter vector for the layer (it contains one offset parameter per input). Each input is offset by its corresponding shift parameter.
- ε is a tiny number that avoids division by zero (typically 10^{-5}). This is called a *smoothing term*.
- $\mathbf{z}^{(i)}$ is the output of the BN operation. It is a rescaled and shifted version of the inputs.

Transfer Learning (to speed-up training)

- One way to speed up the training of deep neural networks is to reuse existing DNNs, which is called transfer learning
- You can get existing DNN models from "model zoos"



- One ongoing challenge is to train a DNN when you don't have enough labeled data for your problem

- If you have a similar, auxiliary problem with large amount of labeled data, then you could train a DNN on this auxiliary problem and then use transfer learning to transfer lower layers of the DNN and then train it with your limited training data

- If you have lot of unlabeled data but not enough labeled ones, then you might generate "bad" records from the good ones and label them bad and the good ones good and train your model to differentiate the two and then use transfer learning to build your model on top of this

- Example for natural language processing:

GOOD		**BAD**
"The dog sleeps"	=>	"The dog they"
"Customer called FB"	=>	"Customer equity FB"

Alternative Optimizers (speed-up)

- Gradient Descent (GD) can be made faster by applying modifications to it to improve the speed of convergence. The update equation for GD, where $J(\boldsymbol{\theta})$ is the loss (cost) function, is

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

  It does not care about what the earlier gradients were. If the local gradient is tiny, it goes very slowly.

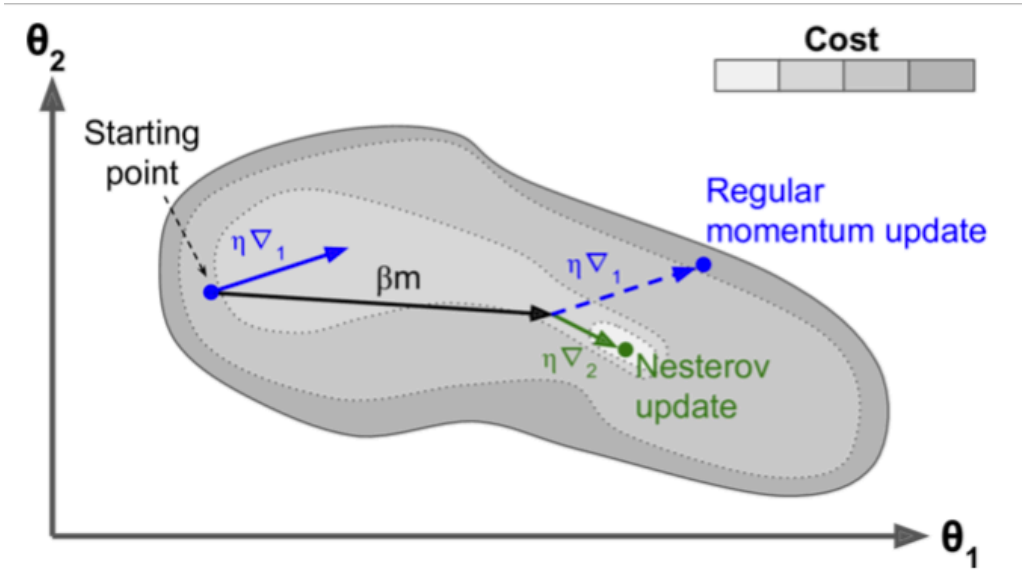
- Momentum Optimization
  - Like a ball rolling down the hill, extend GD with momentum (e.g. the size of next step is influenced by the direction of the previous steps)

$$\boldsymbol{m} \leftarrow \beta \boldsymbol{m} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + m$$

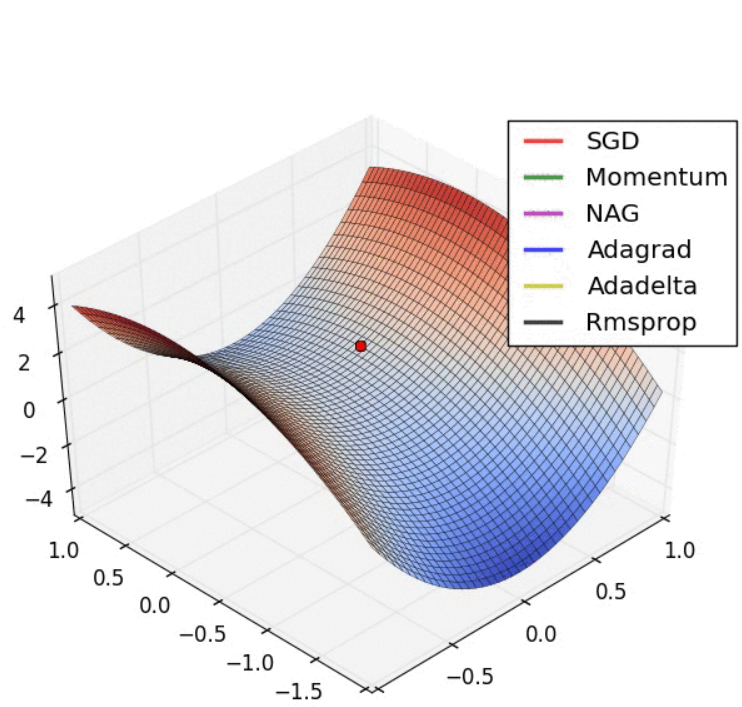
  - If $\beta = 0.9$, the terminal velocity is equal to 10 times ($= 1/(1-\beta)$) the gradient multiplied by the learning rate.

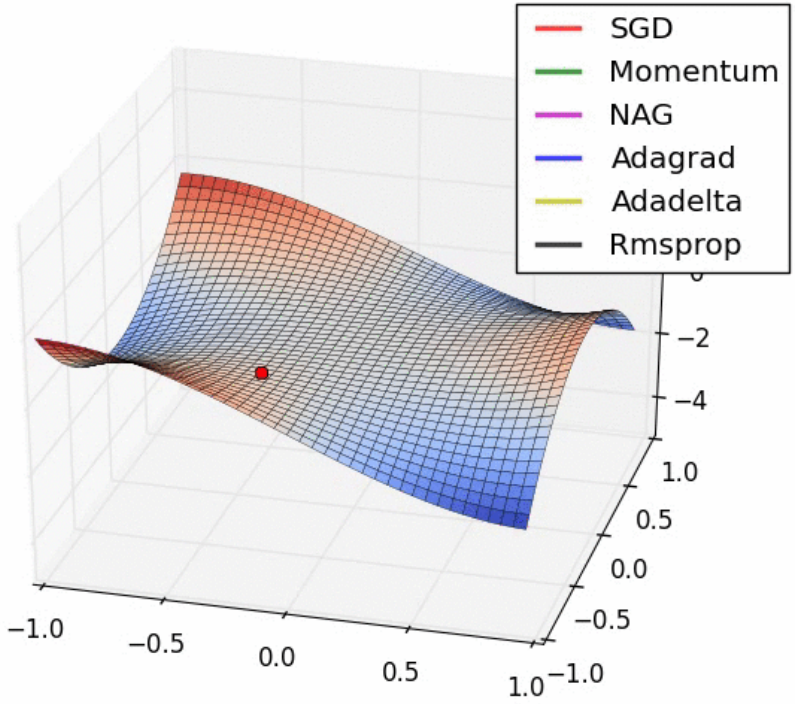
- Nesterov Accelerated Gradient (NAG):
  Variation of the Momentum - measure the gradient not at the local position, but slightly ahead in the direction of the momentum

$$\boldsymbol{m} \leftarrow \beta \boldsymbol{m} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta} + \beta \boldsymbol{m})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + m$$



- Visualize different optimizations

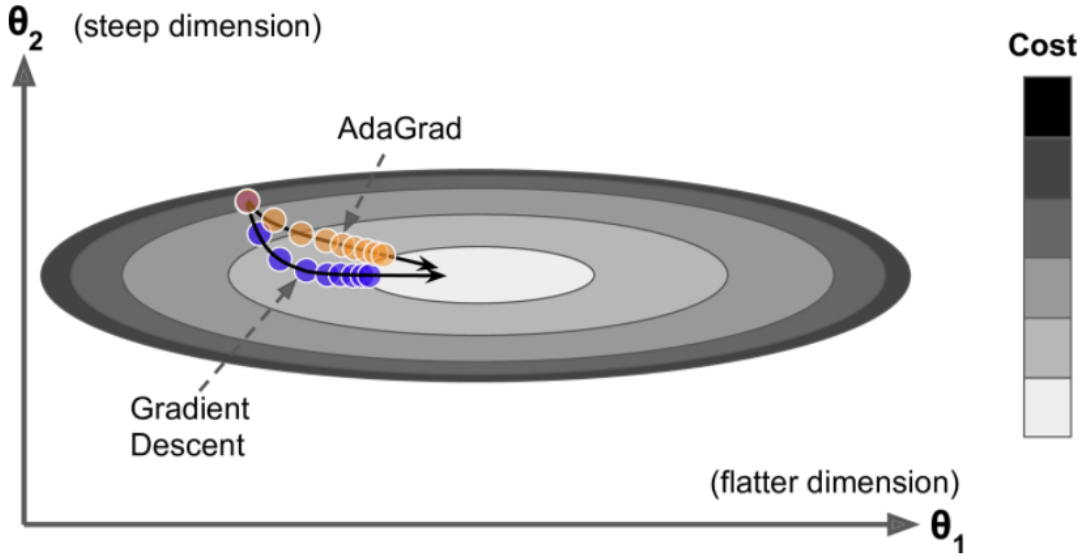




Or this:

https://bl.ocks.org/EmilienDupont/aaf429be5705b219aaaf8d691e27ca87

- AdaGrad (Adaptive Gradient):
  - This algorithm decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes. This is called an adaptive learning rate. It helps point the resulting updates more directly toward the global optimum



  - problem: it might slow down too much and stop before reaching the global minimum

- RMSProp:
  - Similar to AdaGrad, but it doesn't slow down that much, because it only accumulates the gradients from the most recent iterations

- Adam (adaptive momentum estimation) optimization:
  - The combination of the Momentum and the RMSProp
    - just like Momentum optimization it keeps track of an exponentially decaying average of past gradients
    - keeps track of average of past squared gradients (like RMSProp)

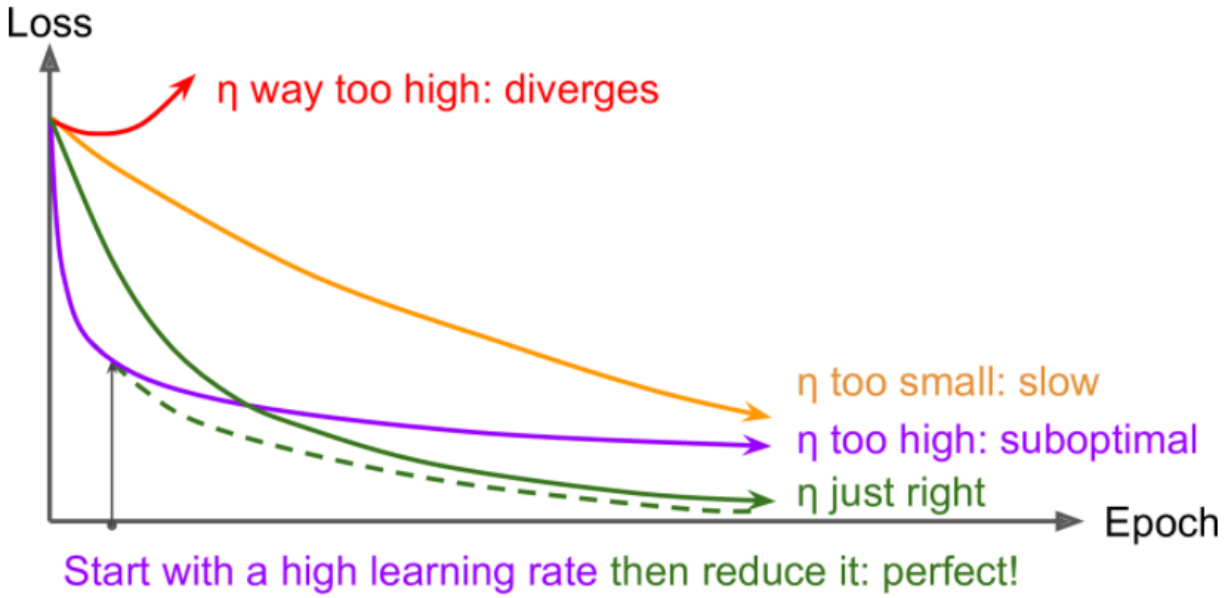
Table 11-2. Optimizer comparison

Class	Convergence speed	Convergence quality
`SGD`	*	***
`SGD(momentum=...)`	**	***
`SGD(momentum=..., nesterov=True)`	**	***
`Adagrad`	***	* (stops too early)
`RMSprop`	***	** or ***
`Adam`	***	** or ***
`Nadam`	***	** or ***
`AdaMax`	***	** or ***

(* is bad, ** is average, and *** is good).

Learning rate is important:

- High learning rate speeds up training, but too high might result in diverging results
- Learning rate scheduling - best of both worlds: start out with a high learning rate and switch to a lower one in the middle of training (unless you use an adaptive learning rate optimization like AdaGrad, RMSProp or Adam)



These are two of the most commonly used learning schedules:

- Power scheduling
  Set the learning rate to a function of the iteration number *t*:

$$\eta(t) = \eta_0 / (1 + t/s)^c.$$

  The initial learning rate η_0, the power c (typically set to 1), and the steps s are hyperparameters. The learning rate drops at each step. When $c = 1$, after s steps it is down to $\eta_0 / 2$. After s more steps, it is down to $\eta_0/3$, then it goes down to $\eta_0/4$, then $\eta_0/5$, and so on. Power scheduling requires tuning η_0 and s (and possibly c).
- Exponential scheduling Set the learning rate to

$$\eta(t) = \eta_0 0.1^{t/s}$$

  The learning rate will gradually drop by a factor of 10 every s steps. While power scheduling reduces the learning rate more and more slowly, exponential scheduling keeps slashing it by a factor of 10 every s steps.

Regularization

- Deep neural networks have tens of thousands of parameters, so it is easy to overfit the training data due to the high number of freedom

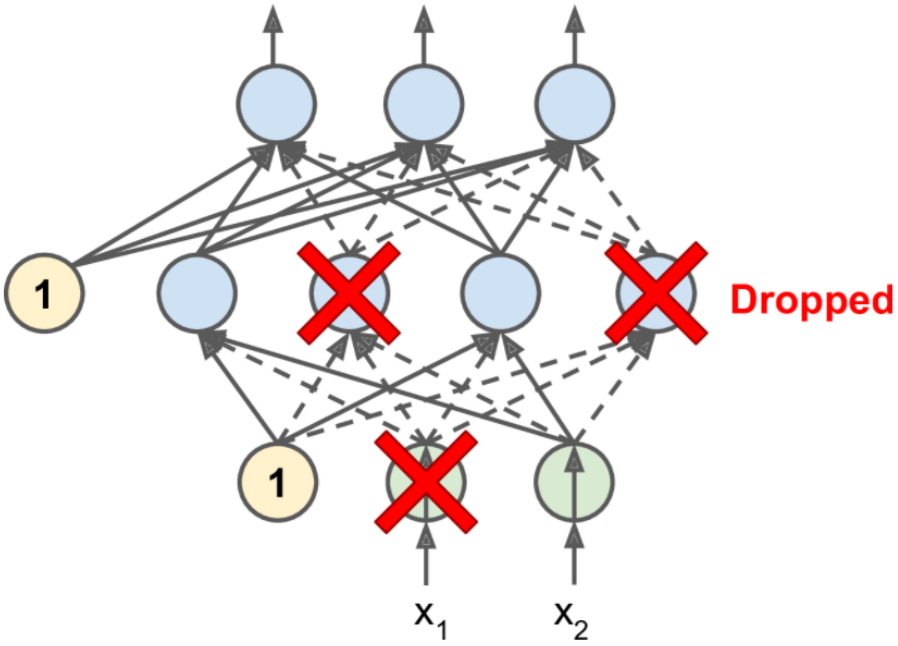
- Regularization is a must have for deep neural networks

- Early Stopping Regularization
  - just interrupt training when its performance on the validation set starts dropping
  - you can usually get much higher performance out of your network by combining it with other regularization techniques

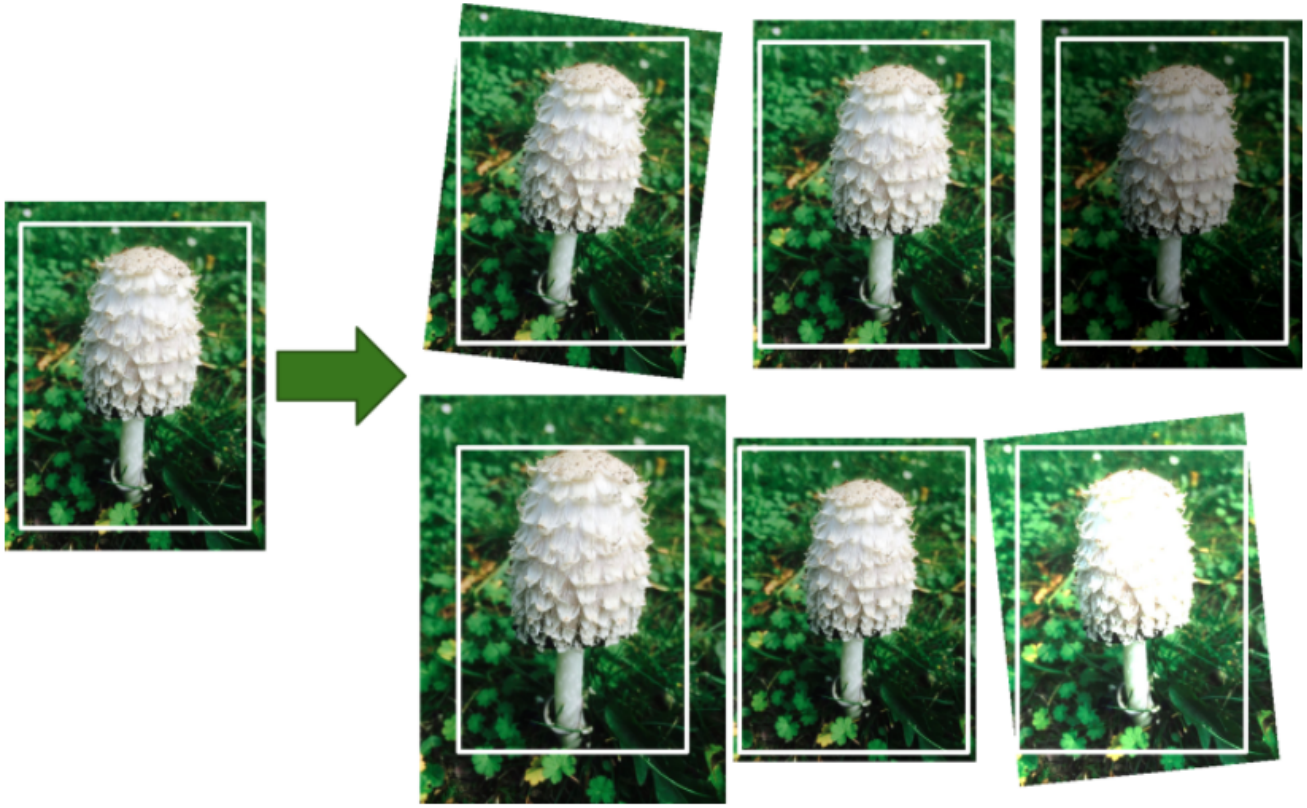
- ℓ1 and ℓ2 Regularization:

It works by adding a summarized form of the weights to the cost function

- Dropout Regularization - the most popular regularization technique introduced in 2012
- Algorithm: at every training step, every neuron (including the input neurons but excluding the output neurons) has a probability p of being temporarily "dropped out," meaning it will be entirely ignored during this training step, but it may be active during the next step



- Data Augmentation - generating new, realistic training instances from existing ones, artificially boosting the size of the training set. This will reduce overfitting - more training data always reduces overfitting.
- Example: mushroom identifier app



State of the art default (2019-20), go-to DNN configuration should probably be:

Hyperparameter	Default value
Kernel initializer	He initialization
Activation function	ELU
Normalization	None if shallow; Batch Norm if deep
Regularization	Early stopping ($+\ell_2$ reg. if needed)
Optimizer	Momentum optimization (or RMSProp or Nadam)
Learning rate schedule	1cycle