BIOS 7747: Machine Learning for Biomedical Applications

Training neural networks

Antonio R. Porras (antonio.porras@cuanschutz.edu)

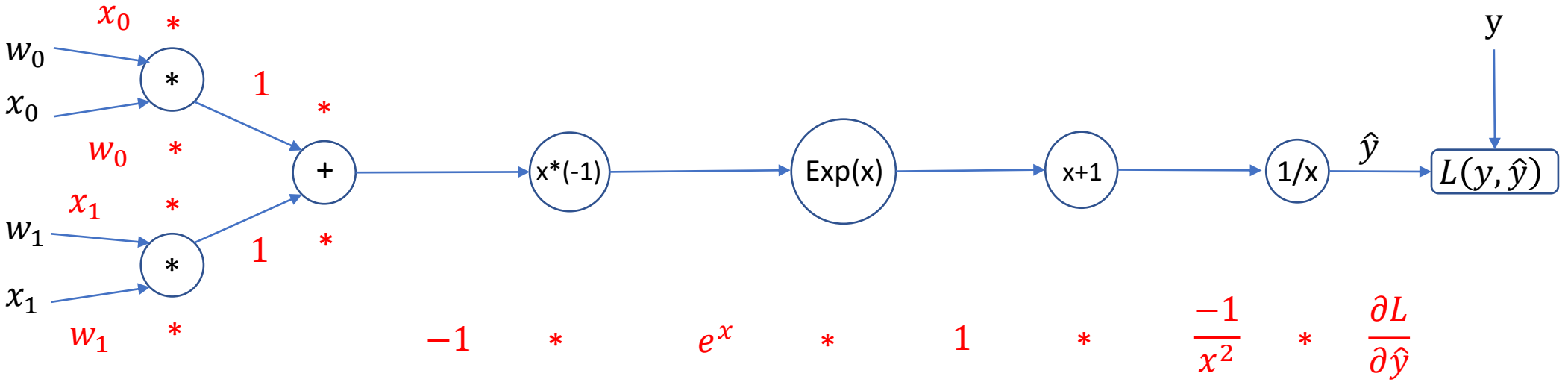
Department of Biostatistics and Informatics
Colorado School of Public Health
University of Colorado Anschutz Medical Campus

Outline

- Vanishing and exploding gradients
  - Activation functions
  - Batch normalization
  - Weight initialization
- Reducing overfitting
  - Regularization
  - Dropout
- Optimization
  - Algorithms
  - Learning rate
  - Stop criteria

Vanishing and exploding gradients

- Let $\hat{y} = f(x; w, b) = \frac{1}{1+\exp(-(w_0 x_0 + w_1 x_1 + b))}$, and $L(y, \hat{y}) = \mathrm{y} - \log(\hat{y}) - (1 - \mathrm{y})\log(1 - \hat{y})$



Situation 1: the gradient of a node is very large → Backpropagated gradients can "explode" and cause numerical instability

Situation 2: the gradient of a node is very small → Backpropagated gradients can "vanish" and prevent optimization

It is very important to keep gradients in optimal ranges

Vanishing and exploding gradients

- Gradients are typically very variable between neurons, which makes it very hard to train deep neural networks
  - Both exploding and vanishing problems can occur during training on different neurons

- To prevent vanishing and exploding gradients:
  - All neurons have input/outputs within similar ranges
  - All neurons have gradients within similar ranges

  } All data should have similar distributions

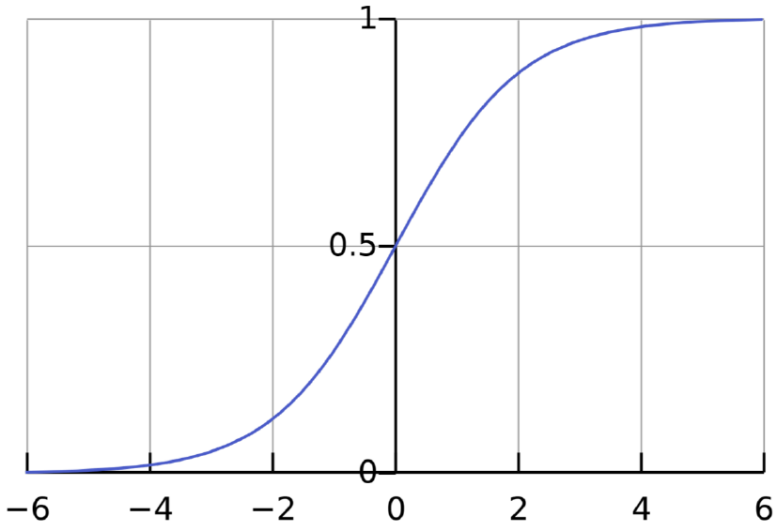
Activation functions

- Sigmoid

In Pytorch
torch.nn.Sigmoid

$$f(x) = \frac{1}{1 + e^{-x}}$$

$$f'(x) = f(x)\big(1 - f(x)\big)$$



- Limited output range: [0, 1]
- Optimal input and output ranges are different
- Zero-gradient problem if x is not in the linear region

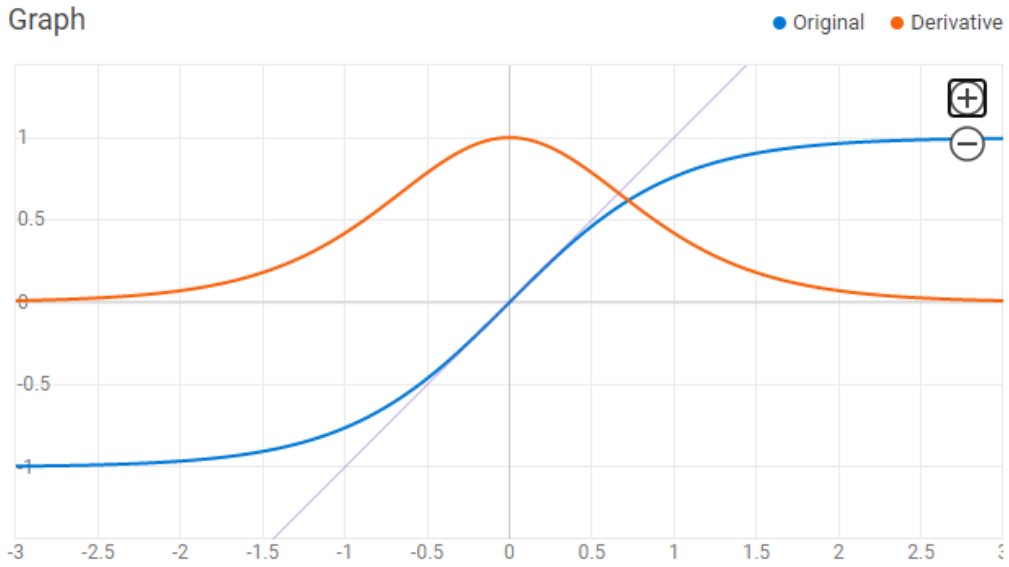
Activation functions

- Tanh (hyperbolic tangent)

In Pytorch
torch.nn.Tanh

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$f'(x) = \operatorname{sech}(x)^2 = \frac{1}{\cosh(x)^2}$$



- Limited output range: [-1, 1]
- Output data are centered around 0
- Zero-gradient problem if x is not in the linear region

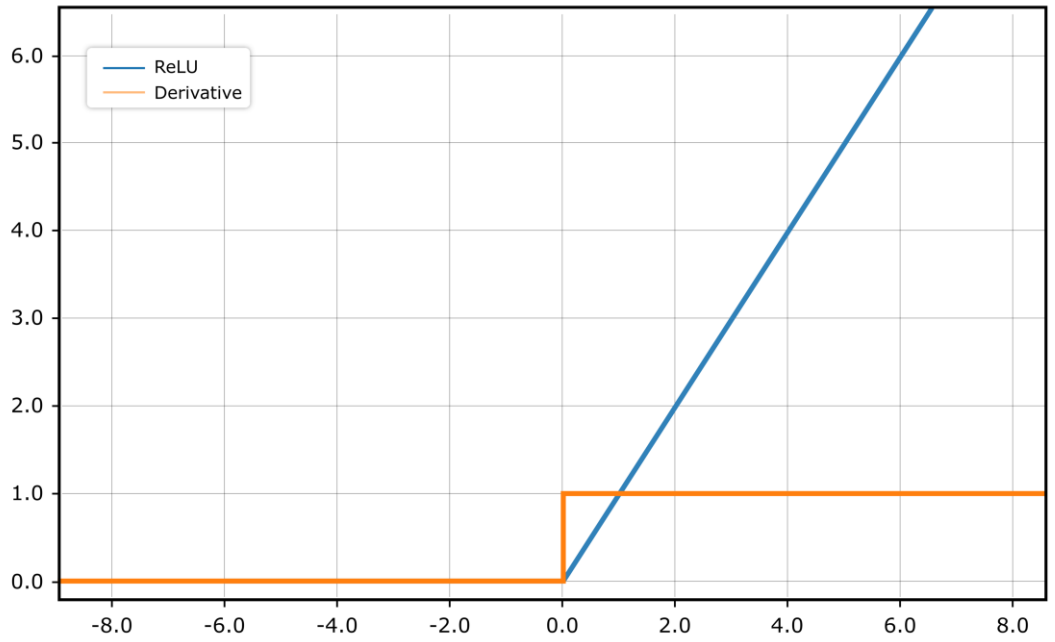
Activation functions

- ReLU

In Pytorch
torch.nn.ReLU

$$f(x) = \max(0, x)$$

$$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{otherwise} \end{cases}$$



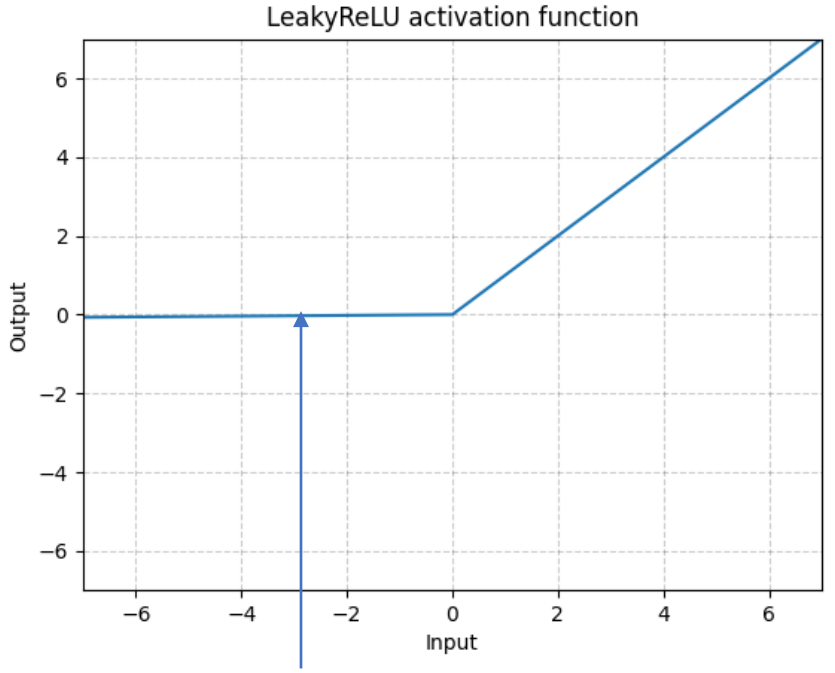
- No gradient issues on activated neurons
- Computationally efficient
- Constant gradient speeds up convergence
- Zero-gradient problem on deactivated neurons

Activation functions

- Leaky ReLU

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0.01x & \text{otherwise} \end{cases}$$

$$f'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0.01 & \text{otherwise} \end{cases}$$



Slight slope

- Solves the problem of “dead” neurons with zero-gradient

In Pytorch

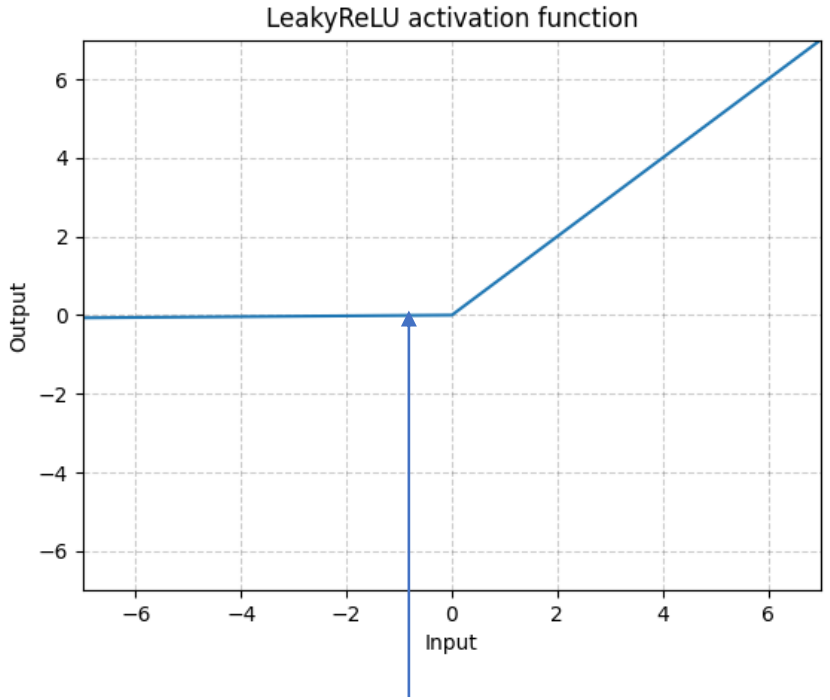
torch.nn.LeakyReLU(negative_slope=0.01)

Activation functions

- Parametric ReLU

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha x & \text{otherwise} \end{cases}$$

$$f'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ \alpha & \text{otherwise} \end{cases}$$



Slope: α

- α can be learned for an "optimal" gradient from deactivated neurons

In Pytorch

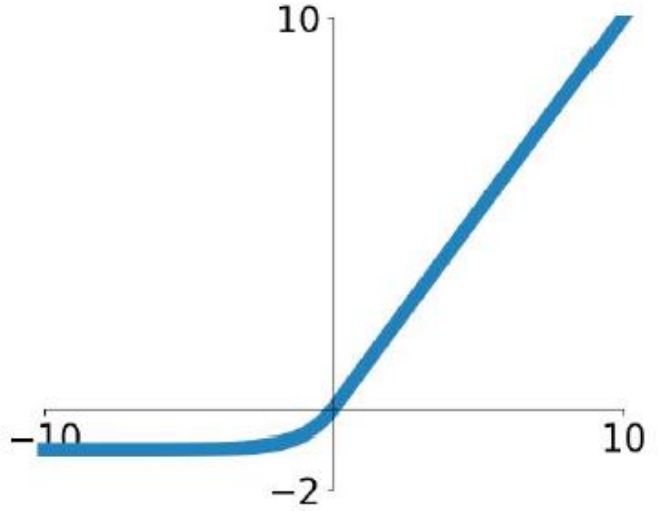
torch.nn.PReLU(num_parameters=1, init=0.25)

Activation functions

- ELU (Exponential linear unit)

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{otherwise} \end{cases}$$

$$f'(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ \alpha e^x & \text{otherwise} \end{cases}$$



- α is a constant (hyperparameter)
- Unlike leaky ReLU, the deactivated values converge to a constant: less prone to overfitting

In Pytorch

torch.nn.ELU(alpha=1.0)

Batch normalization

- The problem of data variability:
  - Operations with inputs or features with high variance will produce outputs of high variance and gradients with high variance
    - Training may be challenging
  - Some activation functions only work well within limited ranges
    - e.g., sigmoid, tanh
  - Other activation functions will not reduce the high variance in the data
    - e.g., ReLU, ELU
  - Batch normalization provides a tool to control the variance in every layer

Batch normalization

- Batch normalization attempts to create feature maps with zero-mean and unit-variance

- If $x = [x^1, x^2, \dots, x^d]$ is a set of activated features in a specific layer, the batch-normalized feature values are:

$$\hat{x}^k = \frac{x^k - E[x^k]}{\text{std}(x^k)}$$

Note: $E[x^k]$ and $\text{std}(x^k)$ are the average and standard deviation of feature x^k in the training dataset

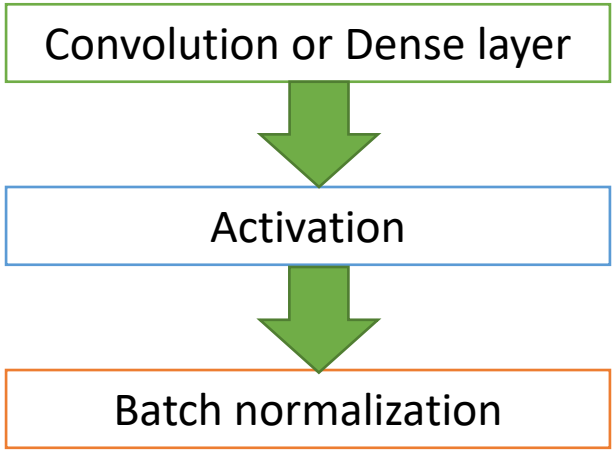
- Running mean and variances are normally used instead of minibatch-only data during training
  - $\text{RunninValue} = \text{momentum} * \text{learnedValue} + (1 - \text{momentum}) * \text{batchValue}$

In Pytorch

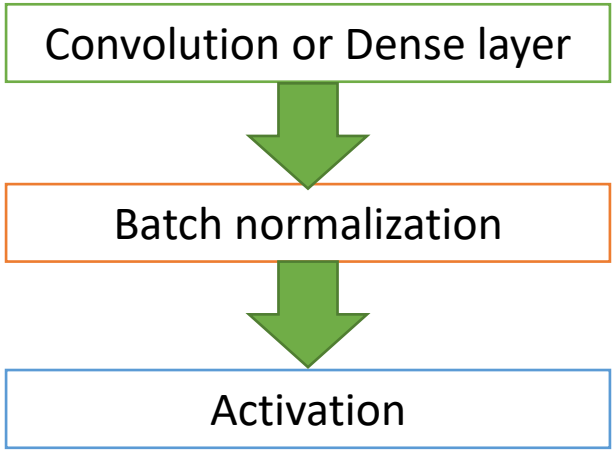
torch.nn.BatchNorm1d(num_features, momentum=0.1)

Ioffe and Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, ICML 2015

Batch normalization



- Useful to limit the activation range (e.g., linear activations)



- Useful to keep non-linear activation functions in the adequate range (e.g., sigmoid, tanh)
- Will also limit the range of linear activations

Batch normalization

- Helps preventing exploding and vanishing gradients by keeping stability in data ranges

- Reduces overfitting by changing feature values during training
  - Adds “noise” during training

- Because of changing values during training, optimization becomes harder

Weight initialization

- Weight initialization

  - Random initialization using normal or uniform distributions

    - If initialized weights are too small, then the variance of the input signal shrinks as it passes through each layer until it's too tiny to be useful [vanishing signal and gradients]

    - If initialized weights are too large, then the variance of the input signal grows as it passes through each layer until it's too large to be useful [exploding signal and gradients]

  - How do we achieve a good weight initialization that preserves data variance?

Weight initialization

- Xavier / Glorot initialization

  - Goal: to keep the same variance after every layer:

$$var(y) = var(w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b)$$

$$var(w_i x_i) = E(x_i)^2 var(w_i) + E(w_i)^2 var(x_i) + var(w_i) var(x_i)$$

Assuming that inputs and weights follow a Gaussian distributions

$$var(w_i x_i) = var(w_i) var(x_i)$$

Assuming zero-mean

$$var(y) = var(w_1) var(x_1) + var(w_2) var(x_2) + \cdots$$

Substituting (b is a constant with zero variance)

$$var(y) = N_{input} * var(w_i) var(x_i)$$

Since all are identically distributed

$$var(w_i) = \frac{1}{N_{input}}$$

Since we want $var(y) = var(x_i)$

Weight initialization

- Xavier / Glorot initialization
  - Normal Xavier initialization that keeps a stable variance in the forward pass:

$$w \in N\left(0, \frac{1}{N_{input}}\right)$$

  - If we want to preserve the variance during backpropagation too:

$$w \in N\left(0, \frac{2}{N_{input} + N_{outputs}}\right)$$

  - Xavier uniform distribution:

$$w \in [-x, x], \qquad x = \sqrt{\frac{6}{N_{input} + N_{outputs}}}$$

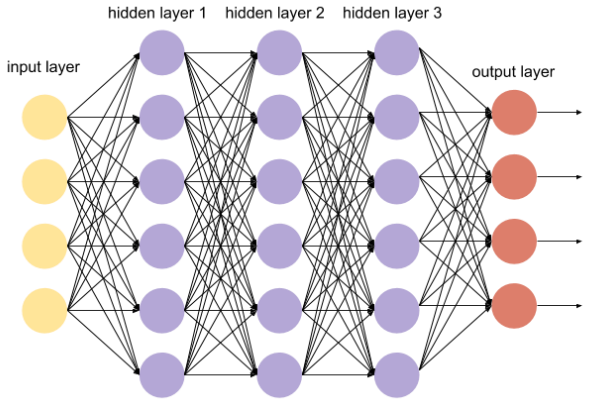
In Pytorch

torch.nn.init.xavier_uniform_() # Initialize weights
torch.nn.init.xavier_normal_() # Initialize weights

Weight initialization

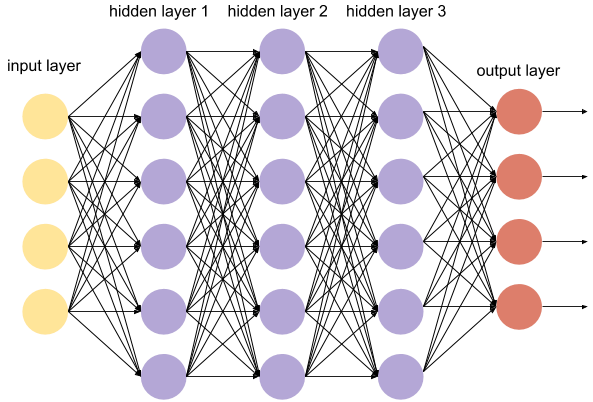
- Transfer learning:
  - Reuse of previously trained weights for a different task

Task 1



Weight transfer

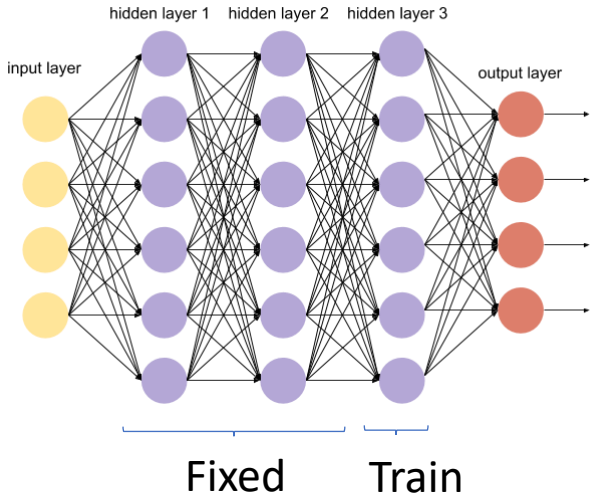
Task 2



Weight initialization

- Transfer learning:
  - Two options after initialization:

Train last layers only

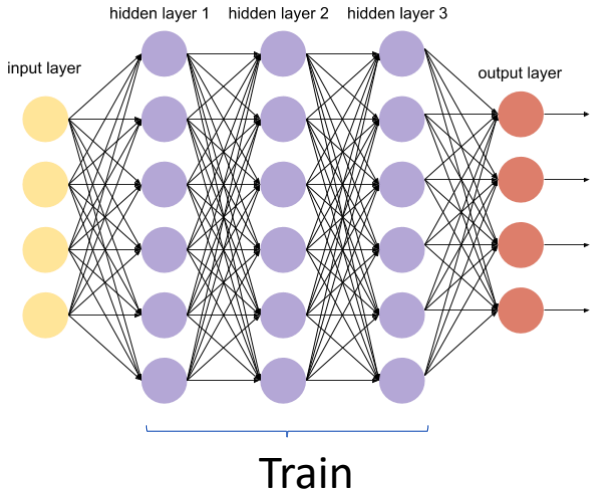


  - Fewer weights to train will reduce overfitting
  - Features in the first layers may not be too relevant for the current problem

Weight initialization

- Transfer learning:
  - Two options after initialization:

Train entire network



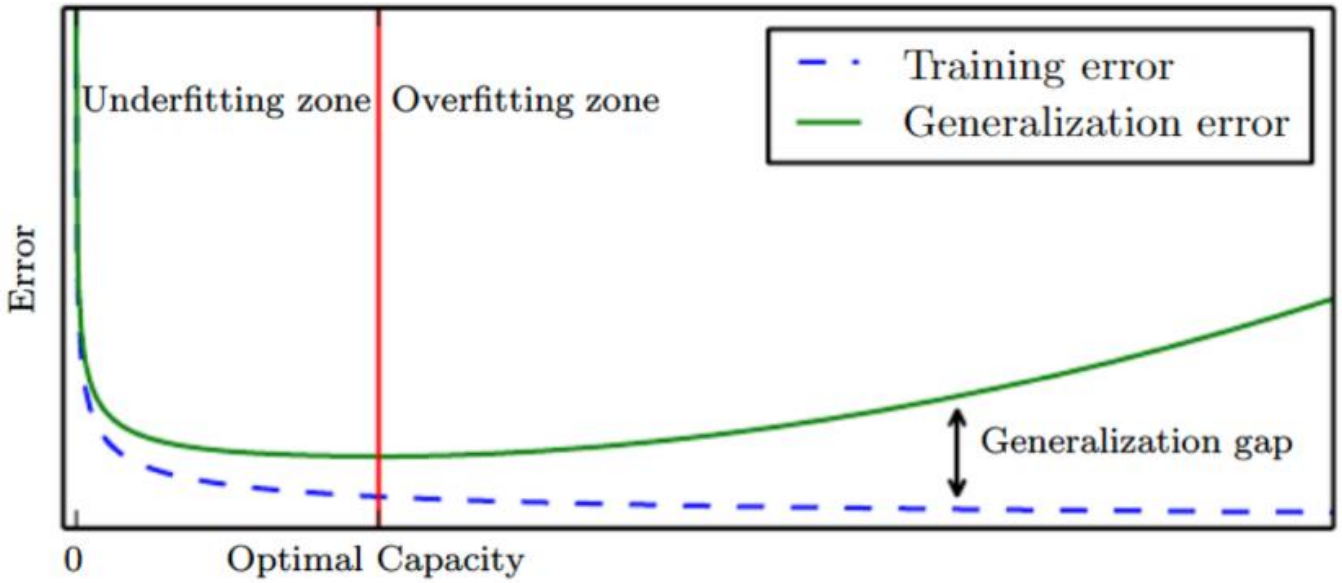
  - More weights to train will be more likely to cause overfitting
  - Features will be problem-specific and may provide higher accuracy

Reducing overfitting: Regularization

- Regularization

  - Goal: to improve the performance of the network on unseen data (generalizability) by reducing overfitting

  - The performance improvement on unseen data usually comes at the expense of a worse training performance



Reducing overfitting: Regularization

- Weight decay:

Regularization weight

$$\arg\min_{W} \frac{1}{N}\sum_{n=1}^{N} L(\hat{y}_n, y_n) + \frac{\lambda}{\beta} R(W)$$

→ Regularization term

Number of weights

- L2 (Ridge) regularization: $R(W) = \sum_{\forall i} w_i^2$ → Keeps weights uniformly small
- L1 (Lasso) regularization: $R(W) = \sum_{\forall i} |w_i|$ → Tends to eliminate weights (deactivate neurons)
- L2 is more commonly used

Reducing overfitting: Regularization

- Weight decay:

  In Pytorch

  L1-normalization

  l1_penalty = torch.nn.L1Loss()
  reg_loss = 0
  for param in model.parameters():
  reg_loss += l1_penalty(param)
  Loss += reg_loss

  L2-normalization

  torch.optim.SGD(…, weight_decay=0)

Reducing overfitting: Dropout

- Dropout

  - Goal: To reduce overfitting by minimizing co-adaptation
  - Co-adaptation: when multiple neurons extract the same or very similar information from the previous layer
  - The problems of co-adaptation:
    - High correlation between extracted features (less meaningful independent information)
    - Extracting the same information in different neurons gives a higher significance to specific features that may only be that significant in the training dataset (promotes overfitting)

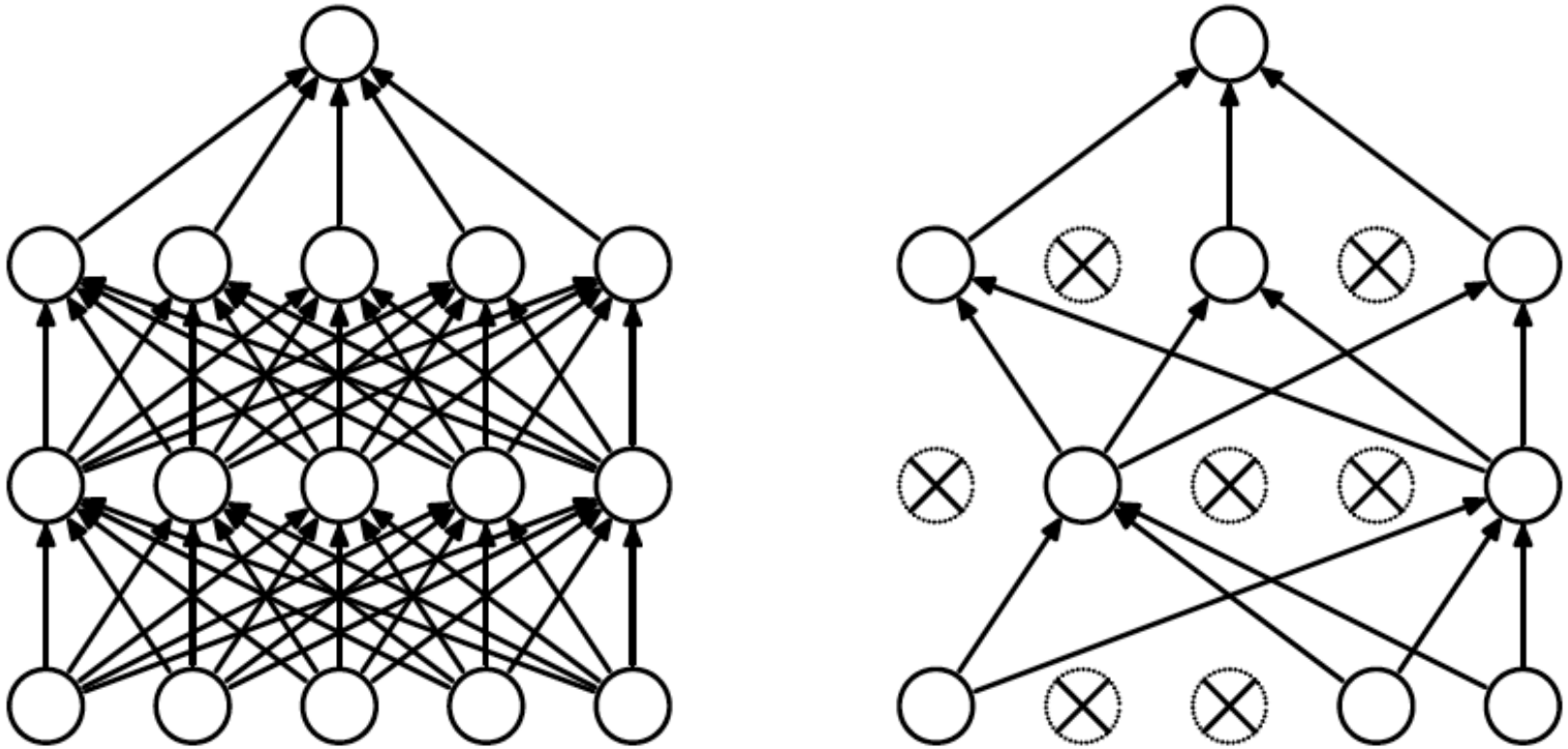
Reducing overfitting: Dropout

- Dropout

  - Dropout consists in eliminating (zeroing) the values of random neurons at each training step
  - Dropout rate (r_D): fraction of neurons deactivated
  - Keep rate: $r_K = 1 - r_D$
  - To preserve data variance, the activated values of the neurons kept is divided by r_K
  - At test time all neurons are preserved

Reducing overfitting: Dropout

- Dropout



(a) Standard Neural Net

(b) After applying dropout.

Srivastava *et al.*, Dropout: A Simple Way to Prevent Neural Networks from Overfitting, *JMLR*, 2014

Reducing overfitting: Dropout

- Dropout

  - When enough training data are available, a dropout rate of 50% is optimal
  - Dropout can make training much more difficult
  - One normally must find a balance between the amount of dropout at each layer, the training and the test performances

In Pytorch

torch.nn.Dropout(p=0.5)

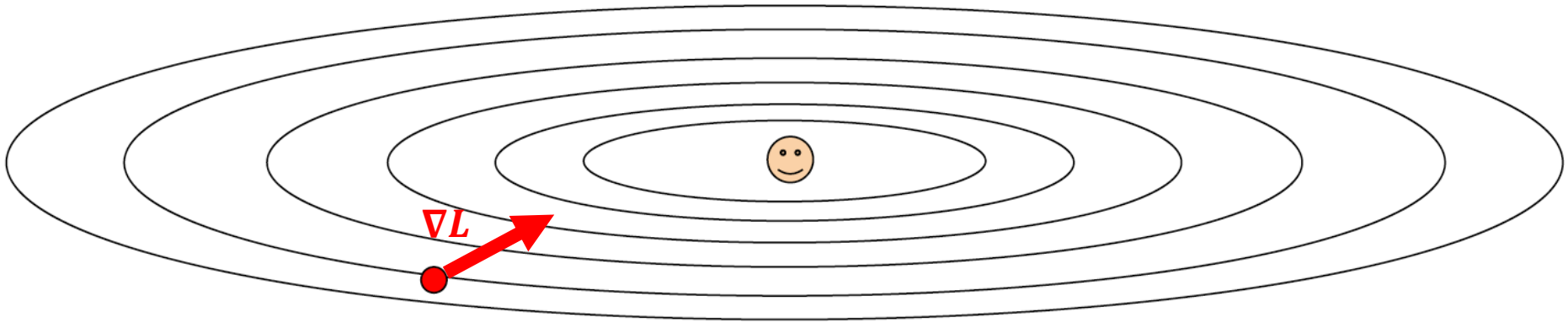
Optimization algorithms

- Regular gradient descent approach:

Learning rate

$$w_t = w_{t-1} - \alpha \nabla L(w_{t-1})$$

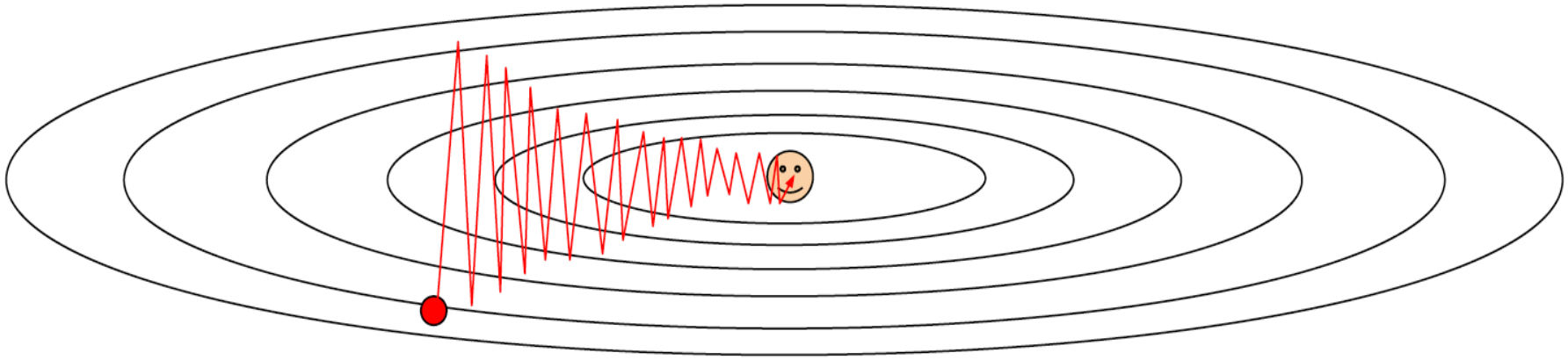
  - May provide good results if gradient estimations are accurate



Optimization algorithms

- Stochastic gradient descent (SGD):

  - Gradients are computed independently for every minibatch and training may be difficult
  - The smaller the minibatch, the harder the training



In Pytorch

torch.optim.SGD(params, lr)

Optimization algorithms

- Stochastic gradient descent (SGD) with momentum:

$$V_t = \beta V_{t-1} + (1 - \beta)\nabla L(W_t)$$

Momentum

$$w_{t+1} = w_t - \alpha V_t$$

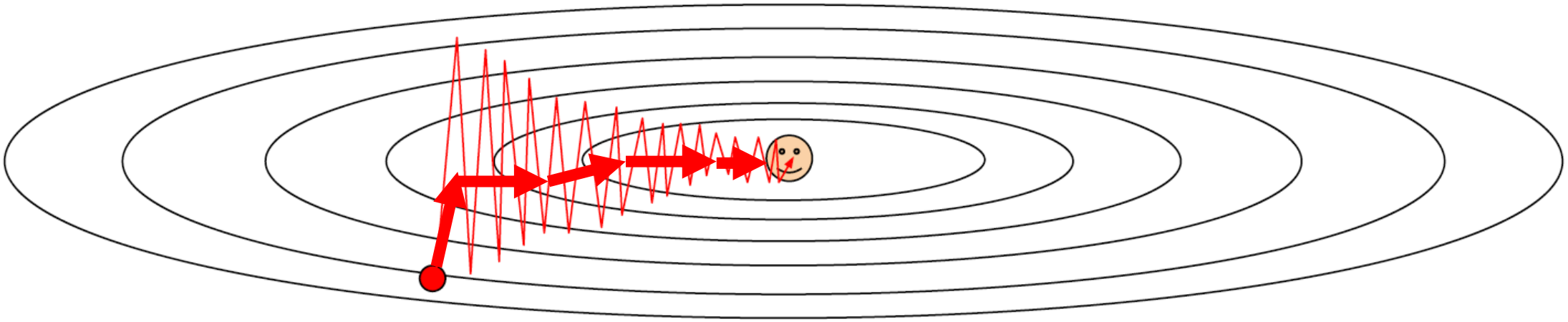
- Higher momentum achieves more stable gradients
- $\beta = 0$ is equivalent to regular SGD
- Typical values ≥ 0.9

In Pytorch

torch.optim.SGD(params, lr, momentum=0)

Optimization algorithms

- Stochastic gradient descent (SGD) with momentum:



Optimization algorithms

- Root mean square propagation (RMSProp):

$$S_t = \beta S_{t-1} + (1-\beta)\left(\nabla L(w_t)\nabla L(w_t)\right)$$

$$w_{t+1} = w_t - \alpha \frac{\nabla L(w_t)}{\sqrt{S_t + \epsilon}}$$

  - Gradients are normalized to a uniform scale that is updated with momentum

In Pytorch β

torch.optim.RMSprop(params, lr=0.01, alpha=0.99)

Optimization algorithms

- Adaptive momentum estimation (Adam):

  - Combines RMSProp and SGD with momentum

$$V_t = \beta_1 V_{t-1} + (1 - \beta_1)\nabla L(w_t)$$

$$S_t = \beta_2 S_{t-1} + (1 - \beta_2)\left(\nabla L(w_t)\nabla L(w_t)\right)$$

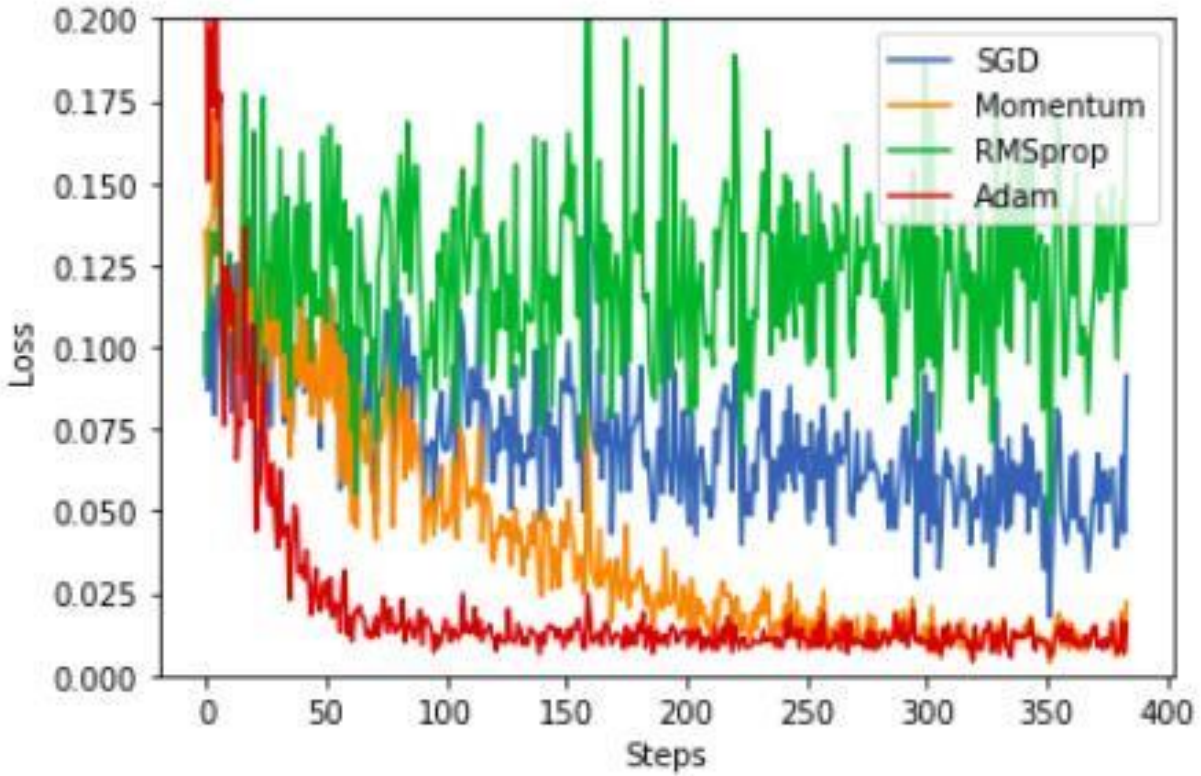
$$w_{t+1} = w_t - \alpha \frac{V_t}{\sqrt{S_t + \epsilon}}$$

In Pytorch

torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999))

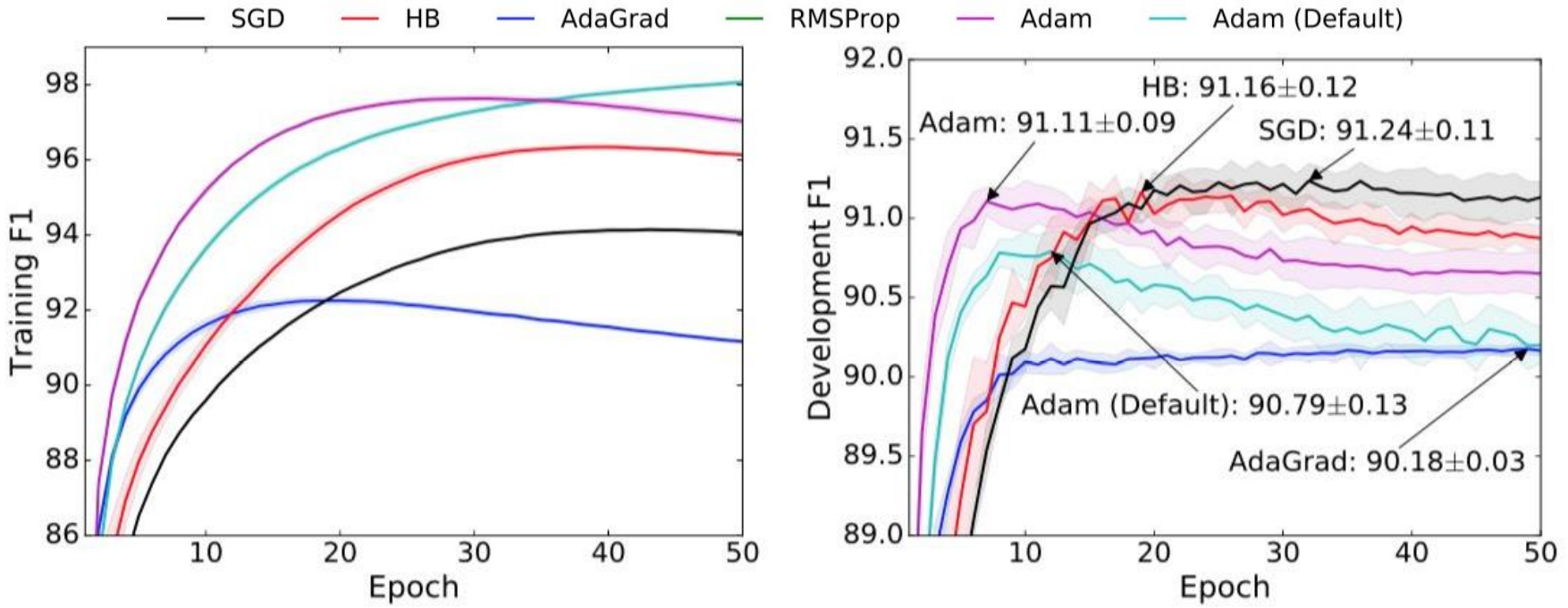
Optimization algorithms

- Example



Optimization algorithms

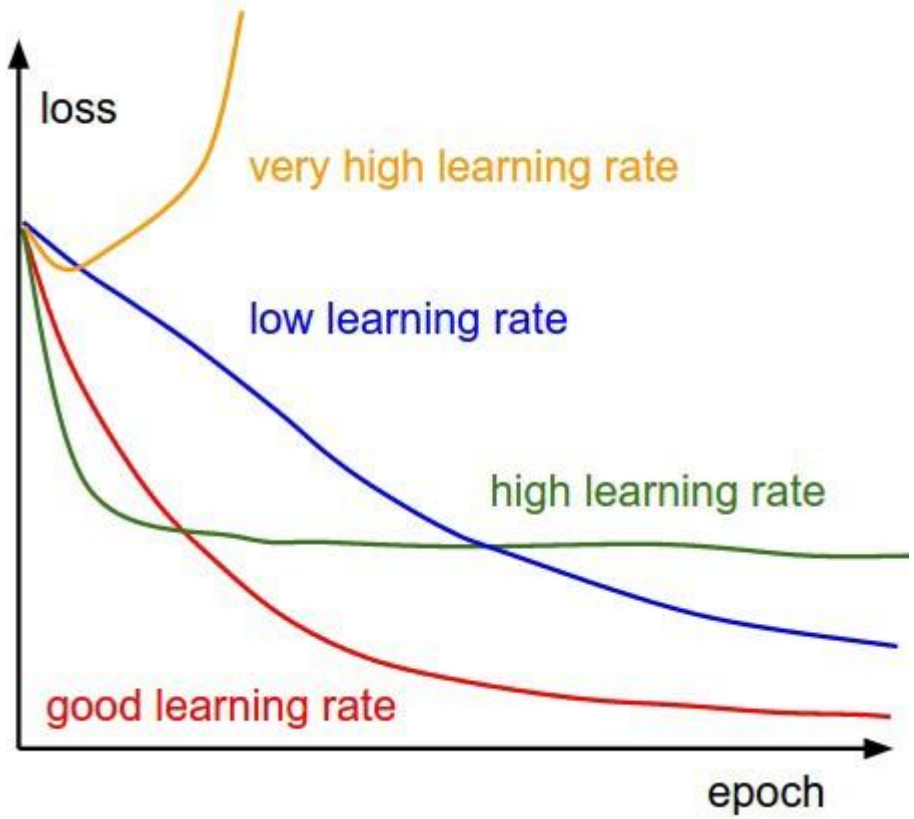
- Thoughts:
  - Lower training loss is not always the best-case scenario
  - One optimizer may provide lower training loss at the expense of higher degree of overfitting
  - Many variables and design decisions affect training and comparisons are difficult (normalization, activation functions, dropout [rate], regularization, etc.)



Optimization algorithms

- Learning rate

  - Hyperparameter that controls how much of the gradient is used to update parameters
  - High values may cause divergence from the optimal solution
  - Low values will likely cause getting trapped in local minima



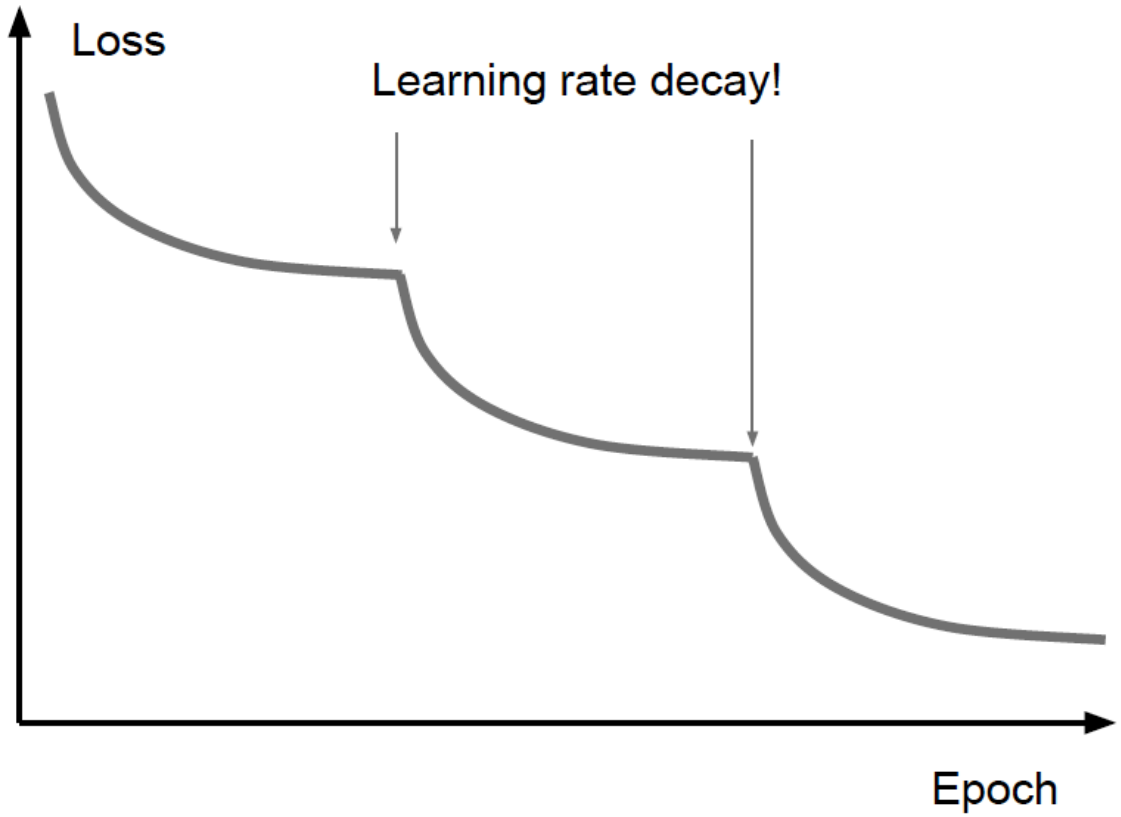
Optimization algorithms

- Learning rate
  - Step decay:
    - Reduction by a fraction every *k* iterations
  - Simple continuous learning rate decay:

$$\alpha_t = \frac{\alpha_0}{1 + kt}$$

  - Exponential decay:

$$\alpha_t = \alpha_0 e^{-kt}$$



Optimization algorithms

- Learning rate

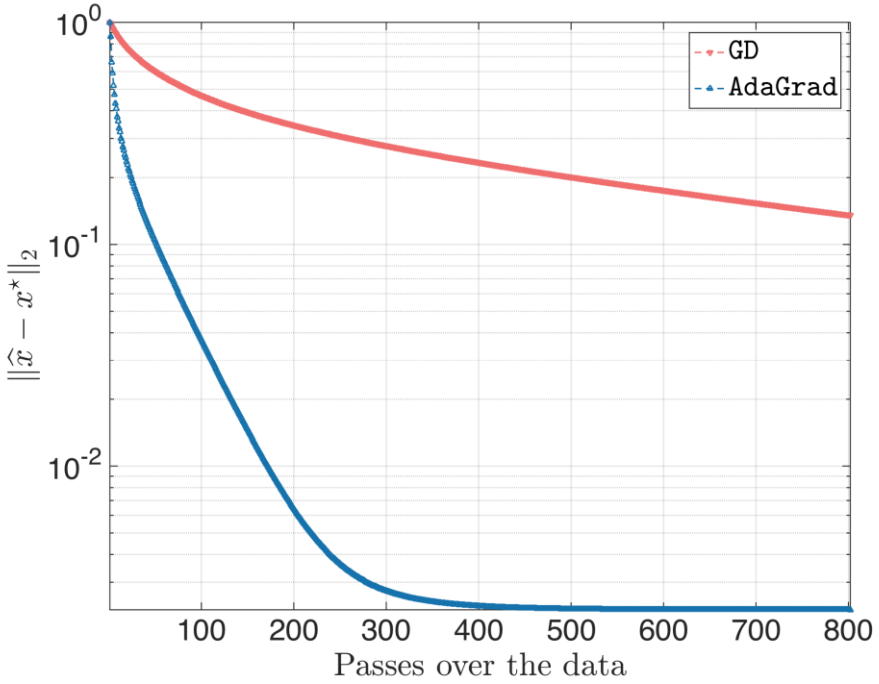
  - Adaptive learning rate: AdaGrad

$$\alpha^{(i)} = \frac{\alpha_0^{(i)}}{\sqrt{\epsilon I + \sum_{\tau=1}^{t} g_\tau^{(i)2}}}$$

Parameter gradient

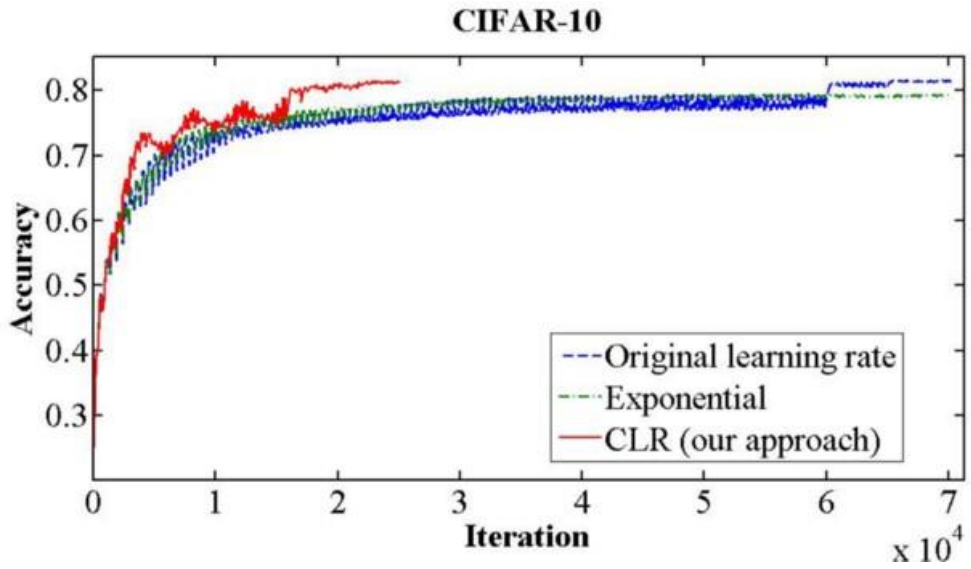
Accounts for total gradient magnitude during optimization

  - There is a different learning rate for each parameter

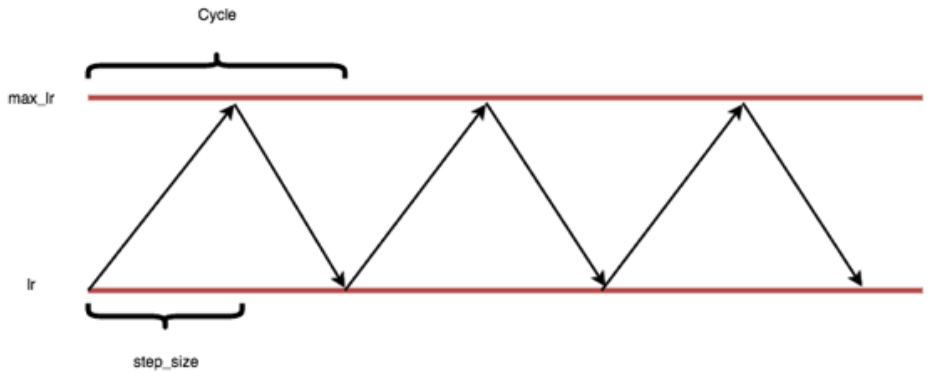


Optimization algorithms

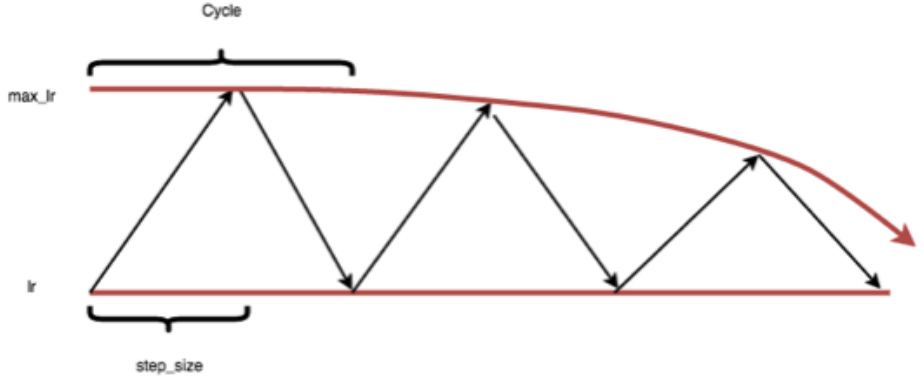
- Learning rate
  - Cyclical learning rate



Basic schedule



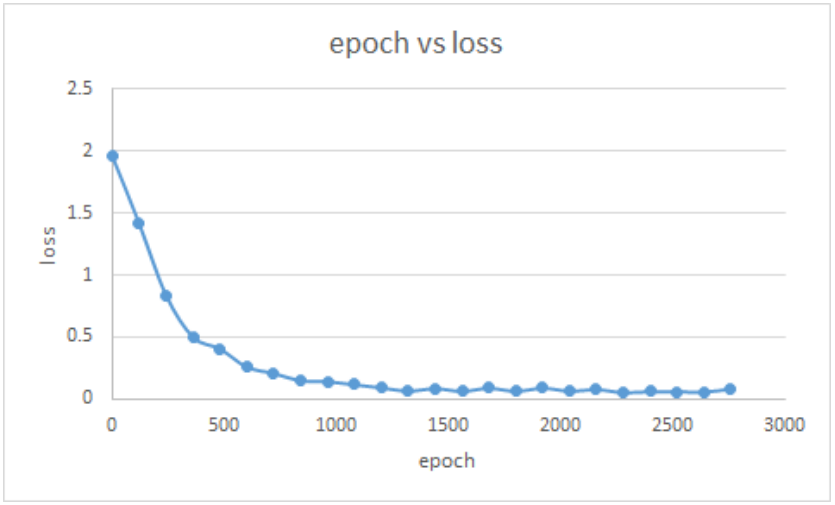
Triangular schedule with exponential decay



He *et al.*, Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, ICCV 2015.

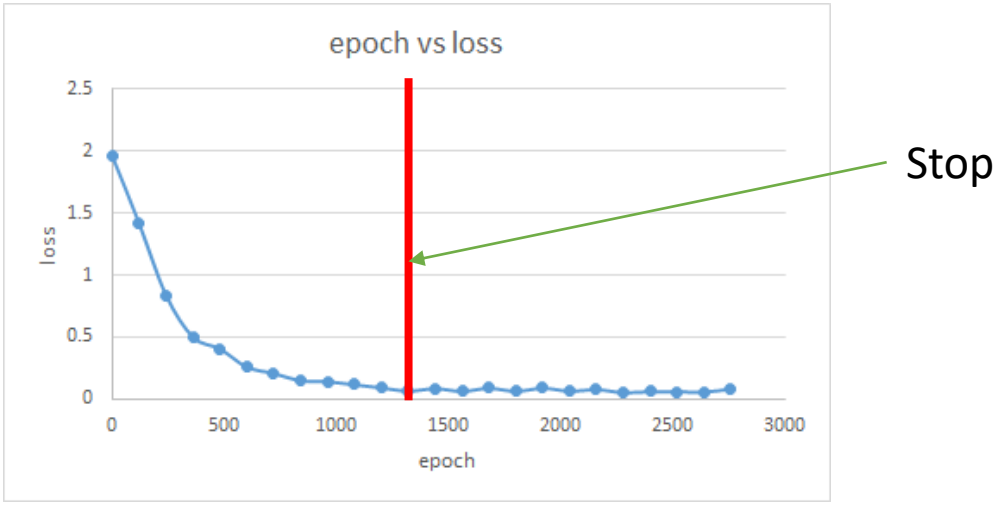
Optimization algorithms

- Stop criterion
  - Fixed number of epochs
    - Problem-, model- and dataset-specific approach
    - Large number can ensure convergence



Optimization algorithms

- Stop criterion
  - Threshold on loss function changes
    - The number of iterations will be different on every model, problem and dataset
    - Can significantly reduce the number of iterations



Optimization algorithms

- Stop criterion
  - Maximum in validation accuracy
    - Requires a representative validation dataset
    - Reduces the size of the dataset used for training (overfitting is more likely) at the expense of having a direct estimation of the amount of overfitting