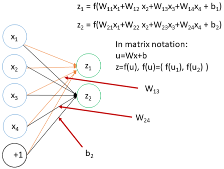
Chapter 4: Introduction to Machine Learning – Optimization, Deep Feed Forward Networks, Backpropagation, Regularization



Christian Heumann
(credits go to Nina Poerner, Benjamin Roth, Marina Speranskaya)

CIS LMU München, Department of Statistics LMU München

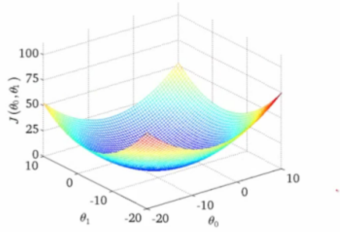
October 2021

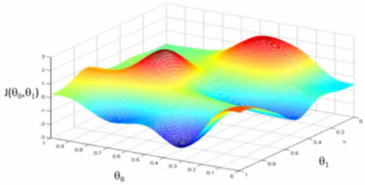
Optimization

- Optimization: Minimize some function $J(\boldsymbol{\theta})$ by altering $\boldsymbol{\theta}$.
- Maximize $f(\boldsymbol{\theta})$ by minimizing $J(\boldsymbol{\theta}) = -f(\boldsymbol{\theta})$
- $J(\boldsymbol{\theta})$:
  - *"criterion"*, *"objective function"*, *"cost function"*, *"loss/risk function"*, *"error function"*
  - In a probabilistic machine learning setting often the (conditional) negative log-likelihood:
    $$-\log p(\boldsymbol{X}; \boldsymbol{\theta})$$
    or
    $$-\log p(\boldsymbol{y}|\boldsymbol{X}; \boldsymbol{\theta})$$
    as a function of $\boldsymbol{\theta}$ is used.
  - $\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$

Optimization

- If $J(\boldsymbol{\theta})$ is convex (figure left), it is minimized where $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbf{0}$
- If $J(\boldsymbol{\theta})$ is not convex (figure right), the gradient can help us to improve our objective nevertheless (and find a local optimum).
- Many optimization techniques were originally developed for convex objective functions, but are found to be working well for non-convex functions too.
- Use the fact that gradient indicates the slope of the function in the direction of steepest increase.

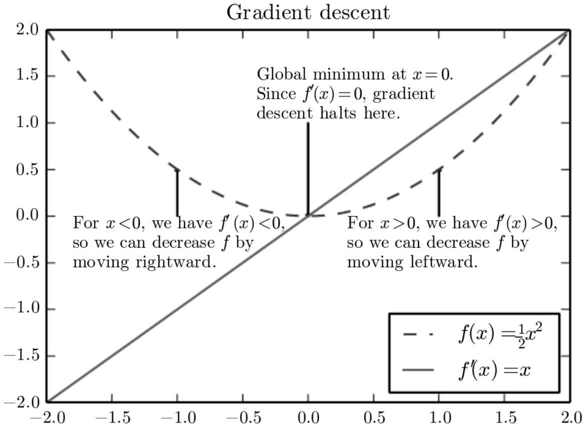




Gradient-Based Optimization

- Derivative: Given a small change in input, what is the corresponding change in output?

$$f(x+\epsilon) \approx f(x) + \epsilon f'(x)$$



- $f(x - \epsilon \,\text{sign}\, f'(x)) < f(x)$ for small enough ϵ

Gradient Descent

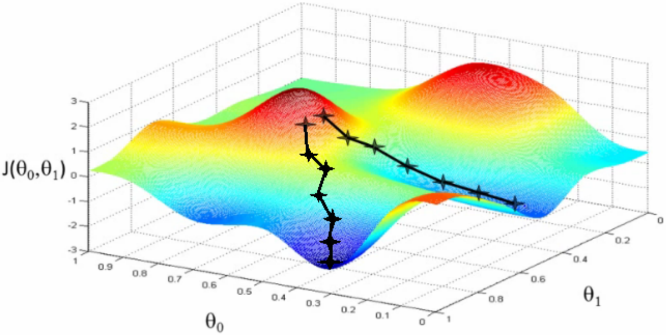
- For $J(\boldsymbol{\theta}) : \mathbb{R}^n \rightarrow \mathbb{R}$
- If partial derivative $\frac{\partial J(\boldsymbol{\theta})}{\partial \theta_j} > 0$, $J(\boldsymbol{\theta})$ will increase for small increases of θ_j
  $\Rightarrow$ go in opposite direction of gradient (since we want to minimize)
- Steepest descent: iterate

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$$

  where $\boldsymbol{\theta}_t$ is the actual parameter, $J(\boldsymbol{\theta}_t)$ is the objective function evaluated at $\boldsymbol{\theta}_t$ and $\boldsymbol{\theta}_{t+1}$ is the updated parameter.
- η is the learning rate (set to a small positive constant).
- Converges if $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is (close to) $\mathbf{0}$

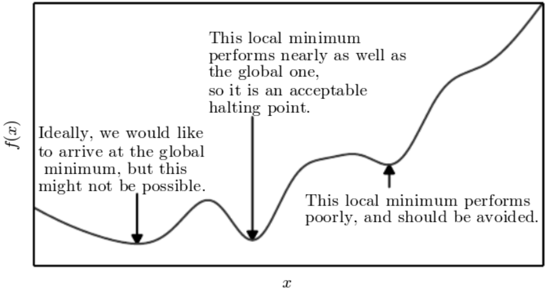
Local Minima

- If the function is non-convex, different results can be obtained at convergence, depending on the initialization of $\boldsymbol{\theta}$ (in DL often: randomly initialized weights).

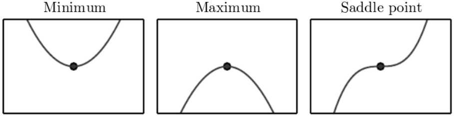


Local Minima

- Minima can be global or local:



- Critical (stationary) points: $f'(x) = 0$



- For (complex) neural networks, only good (not perfect) parameter values can be found.

Gradient Descent for Logistic Regression

$$\nabla_{\boldsymbol{\theta}} NLL(\boldsymbol{\theta}) = -\nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} y^{(i)} \log \sigma(\boldsymbol{\theta}^T \boldsymbol{x}^{(\boldsymbol{i})}) + (1 - y^{(i)}) \log(1 - \sigma(\boldsymbol{\theta}^T \boldsymbol{x}^{(\boldsymbol{i})}))$$

$$= -\sum_{i=1}^{m} (y^{(i)} - \sigma(\boldsymbol{\theta}^T \boldsymbol{x}^{(\boldsymbol{i})}))\boldsymbol{x}^{(\boldsymbol{i})}$$

- The gradient descent update becomes:

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t + \eta \sum_{i=1}^{m} (y^{(i)} - \sigma(\boldsymbol{\theta}_t^T \boldsymbol{x}^{(i)}))\boldsymbol{x}^{(i)}$$

- Note: Which feature weights are increased, which are decreased?

Derivation of Gradient for Logistic Regression

This is a great exercise! Use the following facts:

Gradient	$(\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}))_j = \frac{\partial f(\boldsymbol{\theta})}{\partial \theta_j}$
Derivative of a sum	$\frac{d}{dz} \sum_i f_i(z) = \sum_i \frac{df_i(z)}{dz}$
Chain rule	$F(z) = f(g(z)) \;\Rightarrow\; F'(z) = f'(g(z))g'(z)$
Derivative of logarithm	$\frac{d \log z}{dz} = 1/z$
D. of logistic sigmoid	$\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z))$
Partial d. of dot-product	$\frac{\partial \boldsymbol{\theta}^T \boldsymbol{x}}{\partial \theta_j} = \boldsymbol{x}_j$

Gradient Descent: Summary

- Iterative method for function minimization.
- Gradient indicates the rate of change in the objective function, given a local change to feature weights.
- Substract the gradient:
  - **decrease** parameters that (locally) have **positive** correlation with objective
  - **increase** parameters that (locally) have **negative** correlation with objective
- Gradient updates only have the desired properties in a small region around previous parameters $\boldsymbol{\theta}_t$. Control locality by the learning rate η.
- Gradient descent is slow: For relatively small steps in the right direction, all training data has to be processed.
- This version of gradient descent is often also called *batch gradient descent*.

Stochastic Gradient Descent (SGD)

- *Batch gradient descent* is slow: For relatively small steps in the right direction, all of training data has to be processed.

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \eta \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p(y_i | \boldsymbol{x}_i; \boldsymbol{\theta})$$

- *Stochastic gradient descent* in a nutshell:
  - For each update, only use random sample $\mathbb{B}_t$ of training data (mini-batch).

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \eta \nabla_{\boldsymbol{\theta}} \sum_{i \in \mathbb{B}_t} \log p(y_i | \boldsymbol{x}_i; \boldsymbol{\theta})$$

  - Mini-batch size can also just be 1.

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \eta \nabla_{\boldsymbol{\theta}} \log p(y_t | \boldsymbol{x}_t; \boldsymbol{\theta})$$

- $\Rightarrow$ More frequent updates.

Stochastic Gradient Descent (SGD)

- The actual gradient is *approximated* using only a sub-sample of the data.
- For objective functions that are highly non-convex, the random deviations of these approximations may even help to escape local minima.
- Treat batch size and learning rate as hyper-parameters.

Deep Feedforward Networks

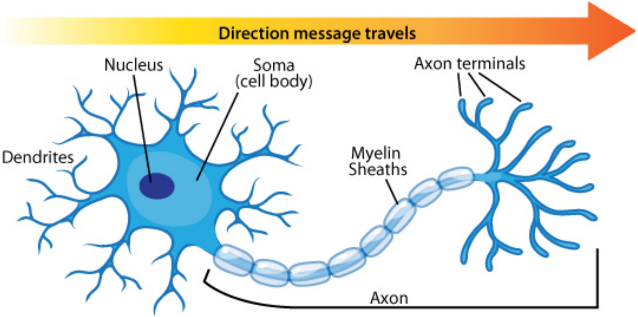
- Function approximation: find good mapping $\hat{\boldsymbol{y}} = f(\boldsymbol{x}; \boldsymbol{\theta})$ (or more exactly $f(\boldsymbol{x}; \hat{\boldsymbol{\theta}})$, but we omit the hat in future).
- *Network*: Composition of functions $f^{(1)}, f^{(2)}, f^{(3)}$ with multi-dimensional input and output
- Each $f^{(i)}$ represents one *layer* $f(\boldsymbol{x}) = f^{(1)}(f^{(2)}(f^{(3)}(\boldsymbol{x})))$
- *Feedforward*:
  - Input $\rightarrow$ intermediate representation $\rightarrow$ output
  - No feedback connections
  - Cf. *recurrent* networks

Deep Feedforward Networks: Training

- Loss function defined on output layer, e.g. $(y - f(\boldsymbol{x}; \boldsymbol{\theta}))^2$
- Quality criterion on other layers not directly defined.
- Training algorithm must decide how to use those layers most effectively (w.r.t. loss on output layer)
- Non-output layers can be viewed as providing a feature function $\phi(\boldsymbol{x})$ of the input, that is to be learned.

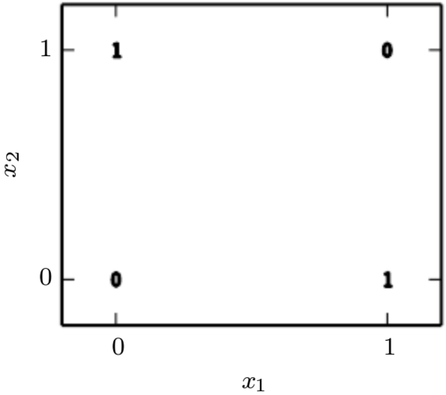
"Neural" Networks

- Inspired by biological neurons (nerve cells)
- Neurons are connected to each other, and receive and send electrical pulses.
- *"If the [input] voltage changes by a large enough amount, an all-or-none electrochemical pulse called an action potential is generated, which travels rapidly along the cell's axon, and activates synaptic connections with other cells when it arrives."* (Wikipedia)



Activation Functions with Non-Linearities

- Linear Functions are limited in what they can express.
- Famous example: XOR
- Simple layered non-linear functions can represent XOR.



Design Choices for Output Units

- Can typically be interpreted as probabilities.
  - Logistic sigmoid
  - Softmax
  - mean and variance of a Gaussian, ...
- Trained with negative log-likelihood.

Softmax

- Logistic sigmoid
  - Vector $\boldsymbol{y}$ of binary outcomes, with no contraints on how many can be 1.
  - Bernoulli distribution.
- Softmax
  - Exactly one element of $\boldsymbol{y}$ is 1.
  - Multinoulli (categorical) distribution.

$$p(Y = i|\phi(\boldsymbol{x}))$$

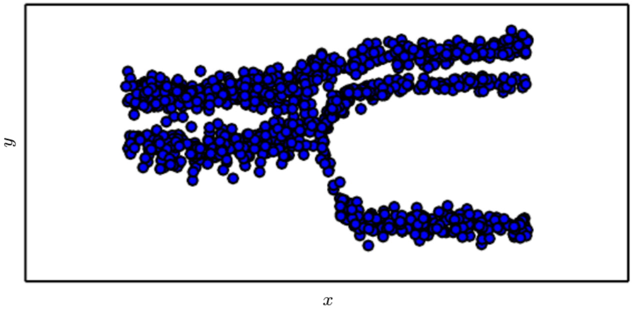
$$\sum_i p(Y = i|\phi(\boldsymbol{x})) = 1$$

$$softmax(\boldsymbol{z})_i = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

Parametrizing a Gaussian Distribution

- Use final layer to predict parameters of Gaussian mixture model.
- Weight of mixture component: softmax.
- Means: no non-linearity.
- Precisions ($\frac{1}{\sigma^2}$) need to be positive: softplus

$$softplus(z) = ln(1 + exp(z))$$

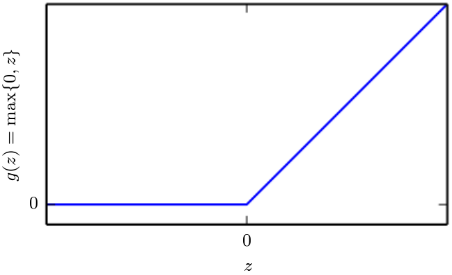


Rectified Linear Units

- Rectified Linear Unit:

$$relu(z) = max(0, z)$$

$$z = \boldsymbol{x}^T \boldsymbol{w} + b$$



- Consistent gradient of 1 when unit is *active* (i.e. if there is an error to propagate).
- Default choice for hidden units.

A Simple ReLU Network to Solve XOR

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}) = \boldsymbol{w}^T max(0, \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{c})$$

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$\boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$\boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$

Other Choices for Hidden Units

- A good activation function aids learning, and provides large gradients.
- Sigmoidal functions (logistic sigmoid)
  - have only a small region before they flatten out in either direction.
  - Practice shows that this seems to be ok in conjunction with Log-loss objective.
  - But they don't work as well as hidden units.
  - ReLU are a better alternative since gradient stays constant.
- Other hidden unit functions:
  - maxout: take maximum of several values in previous layer.
  - purely linear: can serve as low-rank approximation.

- Forward propagation: Input information $\boldsymbol{x}$ propagates through network to produce output $\hat{y}$ (and cost $J(\boldsymbol{\theta})$ in training)
- Back-propagation:
  - compute gradient w.r.t. model parameters
  - Cost gradient propagates backwards through the network
- Back-propagation is part of learning procedure (e.g. stochastic gradient descent), not learning procedure in itself.

Chain Rule of Calculus: Real Functions

- Let

$$x, y, z \in \mathbb{R}$$
$$f, g : \mathbb{R} \to \mathbb{R}$$
$$y = g(x)$$
$$z = f(g(x)) = f(y)$$

- Then

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Chain Rule of Calculus: Multivariate Functions

- Let

$$\boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{y} \in \mathbb{R}^n, z \in \mathbb{R}$$

$$f : \mathbb{R}^n \to \mathbb{R}$$

$$g : \mathbb{R}^m \to \mathbb{R}^n$$

$$\boldsymbol{y} = g(\boldsymbol{x})$$

$$z = f(g(\boldsymbol{x})) = f(\boldsymbol{y})$$

- Then

$$\frac{\partial z}{\partial x_i} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x_i} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x_i} + \ldots + \frac{\partial z}{\partial y_n}\frac{\partial y_n}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

- In order to write this in vector notation, we need to define the Jacobian matrix.

Jacobian

- The Jacobian matrix is the matrix of all first-order partial derivatives of a vector-valued function.

$$\boldsymbol{J} = \frac{\partial \boldsymbol{g}(\boldsymbol{x})}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial g(\boldsymbol{x})_1}{\partial x_1} & \cdots & \frac{\partial g(\boldsymbol{x})_1}{\partial x_m} \\ \frac{\partial g(\boldsymbol{x})_2}{\partial x_1} & & \frac{\partial g(\boldsymbol{x})_2}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial g(\boldsymbol{x})_n}{\partial x_1} & \cdots & \frac{\partial g(\boldsymbol{x})_n}{\partial x_m} \end{bmatrix}$$

- How to write in terms of gradients?
- We can write the chain rule as:

$$\nabla_{\boldsymbol{x}} z =$$

Jacobian

- The Jacobian matrix is the matrix of all first-order partial derivatives of a vector-valued function.

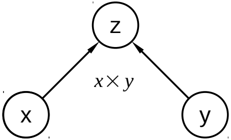
$$\boldsymbol{J} = \frac{\partial \boldsymbol{g}(\boldsymbol{x})}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial g(\boldsymbol{x})_1}{\partial x_1} & \cdots & \frac{\partial g(\boldsymbol{x})_1}{\partial x_m} \\ \frac{\partial g(\boldsymbol{x})_2}{\partial x_1} & & \frac{\partial g(\boldsymbol{x})_2}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial g(\boldsymbol{x})_n}{\partial x_1} & \cdots & \frac{\partial g(\boldsymbol{x})_n}{\partial x_m} \end{bmatrix}$$

- How to write in terms of gradients?
- We can write the chain rule as:

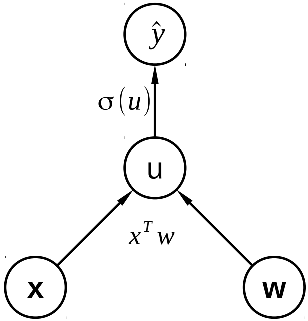
$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \nabla_{\boldsymbol{y}} z$$

Viewing the Network as a Graph

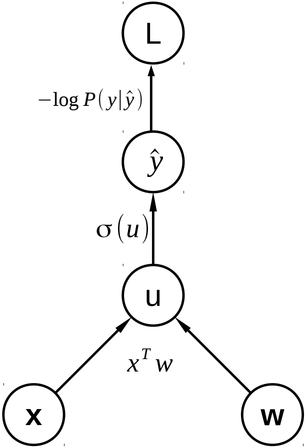
- Nodes are function outputs (can be scalar or vector valued)
- Arrows are inputs
- Example: Scalar multiplication $z = xy$.



Which Function?

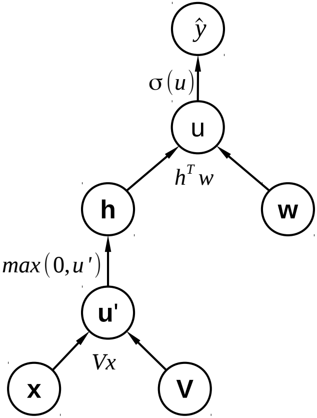


Graph with Cost



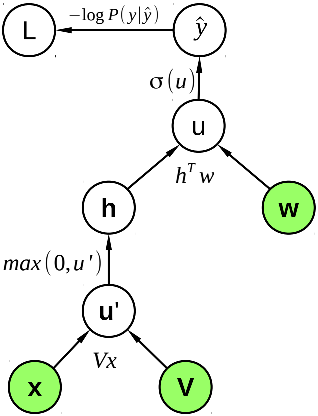
Which Function?

- Parameter vectors can be converted to matrix as needed.



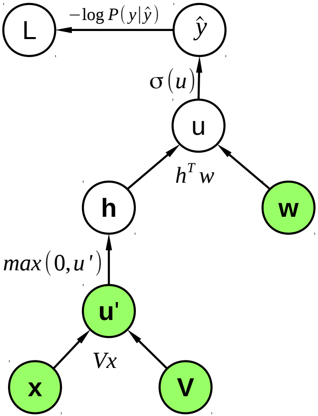
Forward Pass

- Green: known or computed.



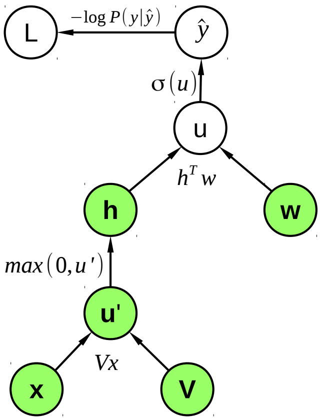
Forward Pass

- Green: known or computed.



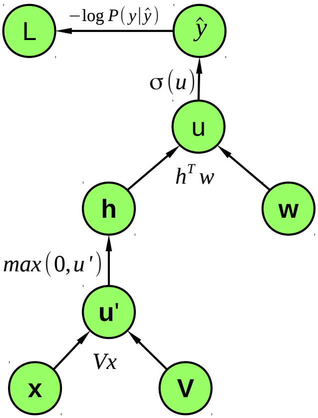
Forward Pass

- Green: known or computed.



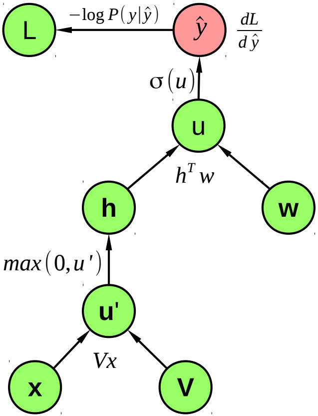
Forward Pass

- End of forward pass (some steps skipped).



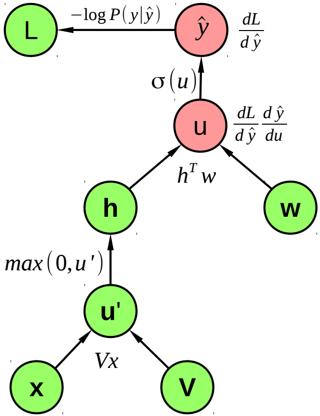
Backward Pass

- Red: gradient of cost computed for node.



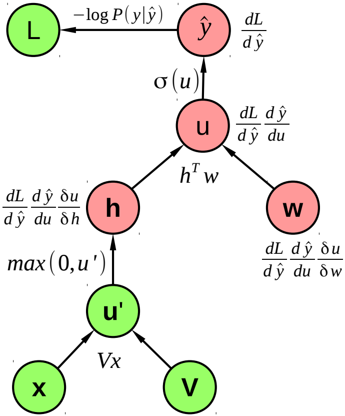
Backward Pass

- Red: gradient of cost computed for node.



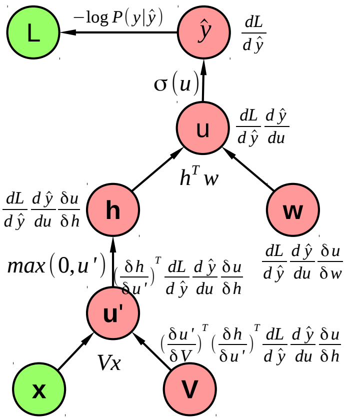
Backward Pass

- Red: gradient of cost computed for node.



End of Backward Pass

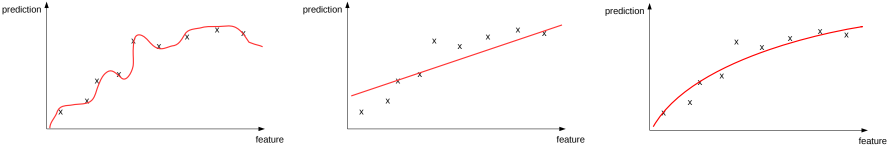
- We have the gradients for all parameters, let's use them for SGD.



Summary

- Gradient descent: Minimize loss by iteratively substracting gradient from parameter vector.
- Stochastic gradient descent: Approximate gradient by considering small subsets of examples.
- Regularization: penalize large parameter values, e.g. by adding L2-norm of parameter vector.
- Feedforward networks: layers of (non-linear) function compositions.
- Output non-linearities: interpreted as probability densities (logistic sigmoid, softmax, Gaussian)
- Hidden layers: Rectified linear units ($max(0, z)$)
- Backpropagation: Compute gradient of cost w.r.t. parameters using chain rule.

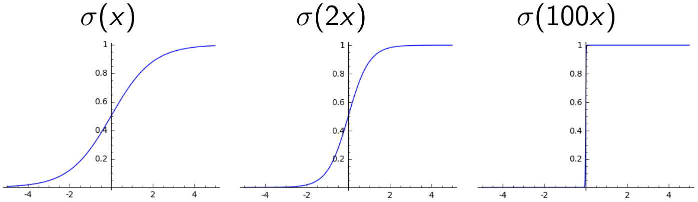
Regularization



- Overfitting vs. underfitting
- Regularization: Any modification to a learning algorithm for reducing its generalization error but not its training error
- Solution space is still the same

L2-Regularization

- Large parameters → overfitting



- Prefer models with smaller feature weights.
- Popular regularizers:
  - Penalize large L2 norm.
  - Penalize large L1 norm (aka LASSO, induces sparsity)

Regularization

- Add term that penalizes large L2 norm.
- The amount of penalty is controlled by a parameter λ
  - Linear regression:

$$J(\boldsymbol{\theta}) = MSE(\boldsymbol{\theta}) + \frac{\lambda}{2}\boldsymbol{\theta}^T\boldsymbol{\theta}$$

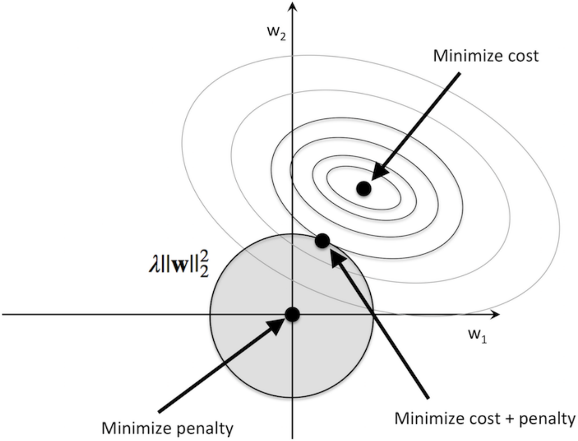
  - Logistic regression:

$$J(\boldsymbol{\theta}) = NLL(\boldsymbol{\theta}) + \frac{\lambda}{2}\boldsymbol{\theta}^T\boldsymbol{\theta}$$

- From a Bayesian perspective, L2-regularization corresponds to a Gaussian prior on the parameters.

L2-Regularization

- The surface of the objective function is now a combination of the original cost, and the regularization penalty.



L2-Regularization

- Gradient of regularization term:

$$\nabla_{\boldsymbol{\theta}} \frac{\lambda}{2} \boldsymbol{\theta}^T \boldsymbol{\theta} = \lambda \boldsymbol{\theta}$$

- Gradient descent for regularized cost function:

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}} (NLL(\boldsymbol{\theta}_t) + \lambda \boldsymbol{\theta}_t^T \boldsymbol{\theta}_t)$$

$$\Leftrightarrow$$

$$\boldsymbol{\theta}_{t+1} := (1 - \eta\lambda) \boldsymbol{\theta}_t - \eta \nabla_{\boldsymbol{\theta}} NLL(\boldsymbol{\theta}_t)$$