

Lecture 2: Modular Learning

Deep Learning @ UvA

Correction for the Piazza bonus

• To make sure there is nothing competitive

• +0.5 bonus to everyone with more than "25 contributions"

• According to the Piazza statistics

• Counting all posts, responses, edits, followups, and comments, there were:

Lecture Overview

- Modularity in Deep Learning
- Popular Deep Learning modules
- Neural Network Cheatsheet
- Backpropagation

The Machine Learning Paradigm

UVA DEEP LEARNING COURSE EFSTRATIOS GAVVES

MODULAR LEARNING - PAGE 4



What is a neural network again?

 A family of parametric, non-linear and hierarchical representation learning functions, which are massively optimized with stochastic gradient descent to encode domain knowledge, i.e. domain invariances, stationarity.

•
$$a^{L}(x; w^{1}, ..., w^{L}) = h^{L}(h^{L-1}(...h^{1}(x, w^{1}), w^{L-1}), w^{L})$$

• x:input, w^{l} : parameters for layer $l, a^{l} = h^{l}(x, w^{l})$: (non-)linear function

• Given training corpus $\{X, Y\}$ find optimal parameters

$$w^* \leftarrow \arg\min_{w} \sum_{(x,y) \subseteq (X,Y)} \mathcal{L}(y, a^L(x))$$

• A neural network model is a series of hierarchically connected functions

• This hierarchies can be very, very complex



Forward connections (Feedforward architecture)

• A neural network model is a series of hierarchically connected functions

• This hierarchies can be very, very complex



• A neural network model is a series of hierarchically connected functions

• This hierarchies can be very, very complex



- A neural network model is a series of hierarchically connected functions
- This hierarchies can be very, very complex



What is a module?

- A module is a building block for our network
- Each module is an object/function a = h(x; w) that
 - Contains trainable parameters w
 - \circ Receives as an argument an input x
 - $^{\circ}$ And returns an output a based on the activation function h(...)
- The activation function should be (at least) first order differentiable (almost) everywhere
- $_{\rm O}$ For easier/more efficient backpropagation \rightarrow store module input
 - easy to get module output fast
 - easy to compute derivatives



Anything goes or do special constraints exist?

- A neural network is a composition of modules (building blocks)
- Any architecture works
- If the architecture is a feedforward cascade, no special care
- If acyclic, there is right order of computing the forward computations
- If there are loops, these form **recurrent** connections (revisited later)

Forward computations for neural networks

• Simply compute the activation of each module in the network

$$a^{l} = h^{l}(x^{l}; w)$$
, where $a^{l} = x^{l+1}$

- \circ We need to know the precise function behind each module $h^l(\dots)$
- Recursive operations
 - One module's output is another's input
- o Steps
 - Visit modules one by one starting from the data input
 - Some modules might have several inputs from multiple modules
- Compute modules activations with the right order
 - Make sure all the inputs computed at the right time



How to get w? Gradient-based learning

• Usually Maximum Likelihood on the training set

$$w^* = \arg\max_{w} \prod_{x,y} p_{model}(y|x;w)$$

• Taking the logarithm, the Maximum Likelihood is equivalent to minimizing the negative log-likelihood cost function

$$\mathcal{L}(w) = -\mathbb{E}_{x, y \sim \tilde{p}_{data}} \log p_{model}(y|x; w)$$

• $p_{model}(y|x)$ is last layer output



If our last layer is the Gaussian function N(y; h(w; x), I) what could be our cost function like? (Multiple answers possible)

• $\sim |y - h(w; x)|^2$ • $\sim \max\{0, 1 - y h(w; x)\}$ • $\sim |y - h(w; x)|_1$ • $\sim |y - h(w; x)|^2 + \lambda \Omega(w)$

How to get w? Gradient-based learning

• Usually Maximum Likelihood in the train set

$$w^* = \arg \max_{\theta} \prod_{x,y} p(y|x;w)$$

• Taking the logarithm, this means minimizing the cost function $\mathcal{L}(\theta) = -\mathbb{E}_{x,y\sim \tilde{p}_{data}} \log p_{model}(y|x;w)$

• $p_{model}(y|x;w)$ is the last layer output $\log p_{model}(y|x) = \log \frac{1}{\sqrt{2\pi\sigma^2}} \exp(\frac{-|y-h(x;w)|^2}{2\sigma^2})$ $\propto C + |y-h(x;w)|^2$ Why should we choose a cost function that matches the form of the last layer of the neural network?

- Otherwise one cannot use standard tools, like automatic differentiation, in packages like Tensorflow or Pytorch
- o It makes the math simpler
- It avoids numerical instabilities
- It makes gradients large by avoiding functions saturating, thus learning is stable

Why should the last network layer "click" with our cost function?

- Otherwise one cannot use standard tools, like automatic differentiation, in packages like Tensorflow or Pytorch
- $\,\circ\,$ It makes the math simpler
- It avoids numerical instabilities
- It makes gradients large by avoiding functions saturating, thus learning is stable

How to get w? Gradient-based learning

• In a neural net $p_{model}(y|x)$ is the module of the last layer (output layer) $\log p_{model}(y|x) = \log \frac{1}{\sqrt{2\pi\sigma^2}} \exp(\frac{-|y-f(\theta;x)|^2}{2\sigma^2}) \Longrightarrow$

$\log p_{model}(y|x) \propto C + |y - f(\theta; x)|^2$

- \circ Everything gets much simpler when the learned (neural network) function p_{model} matches the cost function $\mathcal{L}(w)$
- E.g the log of the negative log-likelihood cancels out the exp of the Gaussian
 - Easier math
 - Better numerical stability
 - Exponential-like activations often lead to saturation, which means gradients are almost 0, which means no learning
- That said, combining any function that is differentiable is possible
 - just not always convenient or smart

Everything is a *module*



MODULAR LEARNING - PAGE 19



- A neural network model is a series of hierarchically connected functions
- This hierarchies can be very, very complex



Linear module

• Activation: a = wx

- Gradient: $\frac{\partial a}{\partial w} = x$
- No activation saturation
- Hence, strong & stable gradients
 - Reliable learning with linear modules



Rectified Linear Unit (ReLU) module

• Activation:
$$a = h(x) = \max(0, x)$$

• Gradient: $\frac{\partial a}{\partial x} = \begin{cases} 0, & \text{if } x \leq 0\\ 1, & \text{if } x > 0 \end{cases}$



What characterizes the Rectified Linear Unit?

- There is the danger the input x is consistently 0 because of a glitch. This would cause "dead neurons" that always are 0 with 0 gradient.
- It is discontinuous, so it might cause numerical errors during training
- It is piece-wise linear, so the "piece"-gradients are stable and strong
- Since they are linear, their gradients can be computed very fast and speed up training.
- They are more complex to implement, because an if condition needs to be introduced.

Rectified Linear Unit (ReLU) module

• Activation:
$$a = h(x) = \max(0, x)$$

- Gradient: $\frac{\partial a}{\partial x} = \begin{cases} 0, & \text{if } x \leq 0\\ 1, & \text{if } x > 0 \end{cases}$
- o Strong gradients: either 0 or 1 $\textcircled{\odot}$
- o Fast gradients: just a binary comparison ☺
- o It is not differentiable at 0, but not a big problem ☺
 - An activation of precisely 0 rarely happens with non-zero weights, and if it happens we choose a convention
- o Dead neurons is an issue
 - Large gradients might cause a neuron to die. Higher learning rates might be beneficial
 - Assuming a linear layer before ReLU $h(x) = \max(0, wx + b)$, make sure the bias term b is initialized with a small initial value, $e. g. 0.1 \rightarrow$ more likely the ReLU is positive and therefore there is non zero gradient
- Nowadays ReLU is the default non-linearity



ReLU convergence rate



Other ReLUs



Softplus



How would you compare the two non-linearities?

- They are equivalent for training
- They are not equivalent for training



Centered non-linearities

- Remember: a deep network is a hierarchy of similar modules
 - One ReLU is the input to the next ReLU
- \circ Consistent behavior \rightarrow input/output distributions must match
 - Otherwise, you will soon have inconsistent behavior
 - If ReLU-1 returns always highly positive numbers, e.g. ~10,000 → the next ReLU-2 biased towards highly positive or highly negative values (depending on the weight w) → ReLU (2) essentially becomes a linear unit.
- We want our non-linearities to be mostly activated around the origin (centered activations)
 - the only way to encourage consistent behavior not matter the architecture

Sigmoid module

• Activation:
$$a = \sigma(x) = \frac{1}{1 + e^{-x}}$$

• Gradient: $\frac{\partial a}{\partial x} = \sigma(x)(1 - \sigma(x))$



Tanh module

• Activation:
$$a = tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

• Gradient: $\frac{\partial a}{\partial x} = 1 - tanh^{2}(x)$



Which non-linearity is better, the sigmoid or the tanh?

- The tanh, because on the average activation case it has stronger gradients
- The sigmoid, because it's output range [0, 1] resembles the range of probability values
- The tanh, because the sigmoid can be rewritten as a tanh
- The sigmoid, because it has a simpler implementation of gradients
- None of them are that great, they saturate for large or small inputs
- The tanh, because it's mean activation is around 0 and it is easier to combine with other modules



UVA DEEP LEARNING COURSE – EFSTRATIOS GAVVES

MODULAR LEARNING - PAGE 31

Which non-linearity is better, the sigmoid or the tanh?

- The tanh, because on the average activation case it has stronger gradients
- The sigmoid, because it's output range [0, 1] resembles the range of probability values
- The tanh, because the sigmoid can be rewritten as a tanh
- The sigmoid, because it has a simpler implementation of gradients
- None of them are that great, they saturate for large or small inputs
- The tanh, because it's mean activation is around 0 and it is easier to combine with other modules



MODULAR LEARNING - PAGE 32

Tanh vs Sigmoids

- Functional form is very similar: $tanh(x) = 2\sigma(2x) 1$
- tanh(x) has better output [-1, +1] range
 - Stronger gradients, because data is centered around 0 (not 0.5)
 - Less "positive" bias to hidden layer neurons as now outputs can be both positive and negative (more likely to have zero mean in the end)
- \circ Both saturate at the extreme \rightarrow 0 gradients
 - "Overconfident", without necessarily being correct
 - Especially bad when in the middle layers: why should a neuron be overconfident, when it represents a latent variable
- The gradients are < 1, so in deep layers the chain rule returns very small total gradient
- From the two, tanh(x) enables better learning
 - But still, not a great choice





Sigmoid: An exception

• An exception for sigmoids is ...

- An exception for sigmoids is when used as the final output layer
- Sigmoid outputs can return very small or very large values (saturate)
 - Output units are not latent variables (have access to ground truth labels)
 - Still "overconfident", but at least towards true values

Softmax module

• Activation:
$$a^{(k)} = softmax(x^{(k)}) = \frac{e^{x^{(k)}}}{\sum_{i} e^{x^{(j)}}}$$

• Outputs probability distribution, $\sum_{k=1}^{K} a^{(k)} = 1$ for K classes

 \circ Avoid exponentianting too large/small numbers \rightarrow better stability

$$a^{(k)} = \frac{e^{x^{(k)} - \mu}}{\sum_{j} e^{x^{(j)} - \mu}}, \mu = \max_{k} x^{(k)} \text{ because}$$
$$\frac{e^{x^{(k)} - \mu}}{\sum_{j} e^{x^{(j)} - \mu}} = \frac{e^{\mu} e^{x^{(k)}}}{e^{\mu} \sum_{j} e^{x^{(j)}}} = \frac{e^{x^{(k)}}}{\sum_{j} e^{x^{(j)}}}$$
Euclidean loss module

• Activation:
$$a(x) = 0.5 ||y - x||^2$$

Mostly used to measure the loss in regression tasks¹⁵

• Gradient:
$$\frac{\partial a}{\partial x} = x - y$$



Cross-entropy loss (log-likelihood) module

(1)

• Activation:
$$a(x) = -\sum_{k=1}^{K} y^{(k)} \log x^{(k)}$$
, $y^{(k)} = \{0, 1\}$

• Gradient:
$$\frac{\partial a}{\partial x^{(k)}} = -\frac{y^{(k)}}{x^{(k)}}$$

- The cross-entropy loss is the most popular classification loss for classifiers that output probabilities
- Cross-entropy loss couples well softmax/sigmoid module
 - The log of the cross-entropy cancels out the exp of the softmax/sigmoid
 - Often the modules are combined and joint gradients are computed
- Generalization of logistic regression for more than 2 outputs

- Everything can be a module, given some ground rules
- How to make our own module?
 - Write a function that follows the ground rules
- Needs to be (at least) first-order differentiable (almost) everywhere
- Hence, we need to be able to compute the

$$\frac{\partial a(x;\theta)}{\partial x}$$
 and $\frac{\partial a(x;\theta)}{\partial \theta}$

- As everything can be a module, a module of modules could also be a module
- We can therefore make new building blocks as we please, if we expect them to be used frequently
- Of course, the same rules for the eligibility of modules still apply

• Assume the sigmoid $\sigma(...)$ operating on top of wx $a = \sigma(wx)$

 \circ Directly computing it \rightarrow complicated backpropagation equations

• Since everything is a module, we can decompose this to 2 modules

$$a_1 = wx \rightarrow a_2 = \sigma(a_1)$$

- Two backpropagation steps instead of one

+ But now our gradients are simpler

- Algorithmic way of computing gradients
- We avoid taking more gradients than needed in a (complex) non-linearity

$$a_1 = wx \rightarrow a_2 = \sigma(a_1)$$

Many, many more modules out there ...

- Many will work comparably to existing ones
 - Not interesting, unless they work consistently better and there is a reason
- Regularization modules
 - Dropout
- Normalization modules
 - ℓ_2 -normalization, ℓ_1 -normalization
- Loss modules
 - Hinge loss

• Most of concepts discussed in the course can be casted as modules

Neural Network Cheatsheet

- Perceptrons, MLPs
- o RNNs, LSTMs, GRUs
- Vanilla, Variational, Denoising Autoencoders
- Hopfield Nets, Restricted Boltzmann Machines
- Convolutional Nets, Deconvolutional Nets
- Generative Adversarial Nets
- Deep Residual Nets, Neural Turing Machines



Backpropagation

UVA DEEP LEARNING COURSE EFSTRATIOS GAVVES & MAX WELLING

MODULAR LEARNING - PAGE 45



- Collect annotated data
- Define model and initialize randomly
- Predict based on current model
 - In neural network jargon "forward propagation"



- Collect annotated data
- Define model and initialize randomly
- Predict based on current model
 - In neural network jargon "forward propagation"



- Collect annotated data
- Define model and initialize randomly
- Predict based on current model
 - In neural network jargon "forward propagation"



- Collect annotated data
- Define model and initialize randomly
- Predict based on current model
 - In neural network jargon "forward propagation"



- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
 - In neural network jargon "backpropagation"



- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
 - In neural network jargon "backpropagation"



- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
 - In neural network jargon "backpropagation"



- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
 - In neural network jargon "backpropagation"



- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
 - In neural network jargon "backpropagation"



Optimization through Gradient Descent

• As for many models, we optimize our neural network with Gradient Descent $w^{(t+1)} = w^{(t)} - \eta \sqrt[r]{v_w}$

- The most important component in this formulation is the gradient
- How to compute the gradients for such a complicated function enclosing other functions, like $a^{L}(...)$?
 - Hint: Backpropagation
- Let's see, first, how to compute gradients with nested functions



• Assume a nested function, z = f(y) and y = g(x)

• When
$$x \in \mathcal{R}^m$$
, $y \in \mathcal{R}^n$, $z \in \mathcal{R}$
• $\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j} \frac{dy_j}{dx_i} \rightarrow$ gradients from all possible paths



• Assume a nested function, z = f(y) and y = g(x)

• When
$$x \in \mathcal{R}^m$$
, $y \in \mathcal{R}^n$, $z \in \mathcal{R}$
• $\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j} \frac{dy_j}{dx_i} \rightarrow$ gradients from all possible paths



$$\frac{dz}{dx^1} = \frac{dz}{dy^1} \frac{dy^1}{dx^1} + \frac{dz}{dy^2} \frac{dy^2}{dx^1}$$

• Assume a nested function, z = f(y) and y = g(x)

• When
$$x \in \mathcal{R}^m$$
, $y \in \mathcal{R}^n$, $z \in \mathcal{R}$
• $\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j} \frac{dy_j}{dx_i} \rightarrow$ gradients from all possible paths



$$\frac{dz}{dx^2} = \frac{dz}{dy^1} \frac{dy^1}{dx^2} + \frac{dz}{dy^2} \frac{dy^2}{dx^2}$$

• Assume a nested function, z = f(y) and y = g(x)

• When
$$x \in \mathcal{R}^m$$
, $y \in \mathcal{R}^n$, $z \in \mathcal{R}$
• $\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j} \frac{dy_j}{dx_i} \rightarrow$ gradients from all possible paths



$$\frac{dz}{dx^3} = \frac{dz}{dy^1} \frac{dy^1}{dx^3} + \frac{dz}{dy^2} \frac{dy^2}{dx^3}$$

• Assume a nested function, z = f(y) and y = g(x)

- Chain Rule for scalars x, y, z• $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$
- O When x ∈ R^m, y ∈ Rⁿ, z ∈ R
 dz/dx_i = ∑_j dz/dy_j dy/dx_i → gradients from all possible paths
 or in vector notation

$$\nabla_{x}(z) = \left(\frac{d\mathbf{y}}{d\mathbf{x}}\right)^{T} \cdot \nabla_{y}(z)$$

• $\frac{dy}{dx}$ is the Jacobian



The Jacobian

• When $x \in \mathcal{R}^3$, $y \in \mathcal{R}^2$

$$J(y(x)) = \frac{dy}{dx} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3} \end{bmatrix}$$

$$\frac{df}{dx} = \frac{d\left[\sin(y)\right]d\left[0.5x^2\right]}{dg}$$
$$= \cos(0.5x^2) \cdot x$$

Backpropagation \Leftrightarrow Chain rule!!!

- The loss function $\mathcal{L}(y, a^L)$ depends on a^L , which depends on a^{L-1} , ..., which depends on a_l
- \circ Gradients of parameters of layer $l \rightarrow$ Chain rule

$$\frac{\partial \mathcal{L}}{\partial w^{l}} = \frac{\partial \mathcal{L}}{\partial a^{L}} \cdot \frac{\partial a^{L}}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial a^{L-2}} \cdot \dots \cdot \frac{\partial a^{l}}{\partial w^{l}}$$

• When shortened, we need to two quantities

$$\frac{\partial \mathcal{L}}{\partial w^{l}} = (\frac{\partial a^{l}}{\partial w^{l}})^{T} \cdot \frac{\partial \mathcal{L}}{\partial a^{l}}$$
Gradient of a module w.r.t. its parameters

Gradient of loss w.r.t. the module output

• For $\frac{\partial a^{l}}{\partial w^{l}}$ in $\frac{\partial \mathcal{L}}{\partial w^{l}} = (\frac{\partial a^{l}}{\partial w^{l}})^{T} \cdot \frac{\partial \mathcal{L}}{\partial a^{l}}$ we only need the Jacobian of the *l*-th module output a^{l} w.r.t. to the module's parameters w^{l}

- <u>Very local</u> rule, every module looks for its own
 - No need to know what other modules do
- Since computations can be very local
 - graphs can be very complicated
 - modules can be complicated (as long as they are differentiable)

Backpropagation \Leftrightarrow Chain rule!!!





How do we compute the gradient of multivariate activation functions, like softmax: $a^{j} = \exp x_{j}/(x_{1} + x_{2} + x_{3})$?

- We vectorize the inputs and the outputs and compute the gradient as before
- We compute the Hessian matrix of the second-order derivatives: $d^2a_j/(dx_idx_j)$
- We compute the Jacobian matrix containing all the partial derivatives: da_j/dx_i



How do we compute the gradient of multivariate activation functions, like softmax: $a^{j} = \exp x_{j}/(x_{1} + x_{2} + x_{3})$?

- We vectorize the inputs and the outputs and compute the gradient as before
- We compute the Hessian matrix of the second-order derivatives: $d^2a_j/(dx_idx_j)$
- We compute the Jacobian matrix containing all the partial derivatives: da_j/dx_i

Multivariate functions f(x)

• Often module functions depend on multiple input variables

• Softmax!

• Each output dimension depends on multiple input dimensions

$$a_j = \frac{e^{x_j}}{e^{x_1} + e^{x_2} + e^{x_3}}, j = 1,2,3$$

 \circ For these cases there are multiple paths for each a_i

So, for the \$\frac{\partial a^l}{\partial x^l}\$ (or \$\frac{\partial a^l}{\partial w^l}\$) we must compute the Jacobian matrix
The Jacobian is the generalization of the gradient for multivariate functions
e.g. in softmax \$a_2\$ depends on all \$e^{x_1}\$, \$e^{x_2}\$ and \$e^{x_3}\$, not just on \$e^{x_2}\$

• But, quite often in modules the output depends only in a single input • e.g. a sigmoid $a = \sigma(x)$, or a = tanh(x), or a = exp(x)

$$a(x) = \sigma(\mathbf{x}) = \sigma\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} \sigma(x_1) \\ \sigma(x_2) \\ \sigma(x_3) \end{bmatrix}$$

• Not need for full Jacobian, only the diagonal: anyways $\frac{da_i}{dx_j} = 0$, $\forall i \neq j$

$$\frac{d\boldsymbol{a}}{d\boldsymbol{x}} = \frac{d\boldsymbol{\sigma}}{d\boldsymbol{x}} = \begin{bmatrix} \sigma(x_1)(1 - \sigma(x_1)) & 0 & 0 \\ 0 & \sigma(x_2)(1 - \sigma(x_2)) & 0 \\ 0 & 0 & \sigma(x_3)(1 - \sigma(x_3)) \end{bmatrix} \sim \begin{bmatrix} \sigma(x_1)(1 - \sigma(x_1)) \\ \sigma(x_2)(1 - \sigma(x_2)) \\ \sigma(x_3)(1 - \sigma(x_3)) \end{bmatrix}$$

• Can rewrite equations as inner products to save computations

Forward graph

• Simply compute the activation of each module in the network

$$a^{l} = h^{l}(x^{l}; w)$$
, where $a^{l} = x^{l+1}$ (or $x^{l} = a^{l-1}$)

• We need to know the precise function behind each module $h^l(...)$

• Recursive operations

• One module's output is another's input

• Store intermediate values to save time

o Steps

• Visit modules one by one starting from the data input

- Some modules might have several inputs from multiple modules
- Compute modules activations with the right order
 - Make sure all the inputs computed at the right time



- Simply compute the gradients of each module for our data
 - We need to know the gradient formulation of each module $\partial h^l(x^l; w^l)$ w.r.t. their inputs x^l and parameters w^l
- We need the **forward computations first**
 - Their result is the sum of losses for our input data
- Then take the reverse network (reverse connections) and traverse it backwards
- Instead of using the activation functions, we use their gradients
- The whole process can be described very neatly and concisely with the backpropagation algorithm



Backpropagation: Recursive chain rule

• Step 1. Compute forward propagations for all layers recursively

$$a^l = h^l(x^l)$$
 and $x^{l+1} = a^l$

Step 2. Once done with forward propagation, follow the reverse path.
 Start from the last layer and for each new layer compute the gradients

• Cache computations when possible to avoid redundant operations



• Step 3. Use the gradients $\frac{\partial \mathcal{L}}{\partial w^l}$ with Stochastic Gradient Descend to train
Backpropagation: Recursive chain rule

• Step 1. Compute forward propagations for all layers recursively

$$a_l = h_l(x_l)$$
 and $x_{l+1} = a_l$

Step 2. Once done with forward propagation, follow the reverse path. Start from the last layer and for each new layer compute the gradients Vector with dimensions $[d_{l-1} \times 1]$ Cache computations when possible to avoid redundant operations Vector with dimensions $\begin{bmatrix} d_l \times 1 \end{bmatrix}$ $\frac{\partial \mathcal{L}}{\partial a_l} = \left(\frac{\partial a_{l+1}}{\partial x_{l+1}}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a_{l+1}}$ $\left|\frac{\partial \mathcal{L}}{\partial \theta_{l}} = \frac{\partial a_{l}}{\partial \theta_{l}} \cdot \right|$ $\partial \mathcal{L} \setminus^T$ • Step 3. Use the gradients $\frac{\partial \mathcal{L}}{\partial \theta_1}$ with Stochastic Gradient Descend to train Vector with dimensions $[1 \times d_I]$ Jacobian matrix with dimensions $[d_{l+1} \times d_l]^T$ Matrix with dimensions $[d_{l-1} \times d_l]$ Vector with dimensions $[d_{l+1} \times 1]$

Backpropagation visualization



Forward propagations

Compute and store $a_1 = h_1(x_1)$



Forward propagations

Compute and store $a_2 = h_2(x_2)$



Forward propagations

Compute and store $a_3 = h_3(x_3)$







UVA DEEP LEARNING COURSE – EFSTRATIOS GAVVES

Backpropagation

 $\frac{\partial \mathcal{L}}{\partial a_1} = \frac{\partial \mathcal{L}}{\partial a_2} \cdot \frac{\partial a_2}{\partial a_1}$ $\frac{\partial \mathcal{L}}{\partial \theta_1} = \frac{\partial \mathcal{L}}{\partial a_1} \cdot \frac{\partial a_1}{\partial \theta_1}$



Forward propagations

Compute and store $a_1 = h_1(x_1)$



Forward propagations

Compute and store $a_2 = h_2(x_2)$



Forward propagations

Compute and store $a_3 = h_3(x_3)$





Backpropagation $a \int a \int a a^3$

 $\frac{\partial \mathcal{L}}{\partial a^2} = \frac{\partial \mathcal{L}}{\partial a^3} \cdot \frac{\partial a^3}{\partial a^2}$ $\frac{\partial \mathcal{L}}{\partial w^2} = \frac{\partial \mathcal{L}}{\partial a^2} \cdot \frac{\partial a^2}{\partial w^2}$



Backpropagation

 $\frac{\partial \mathcal{L}}{\partial a_1} = \frac{\partial \mathcal{L}}{\partial a_2} \cdot \frac{\partial a_2}{\partial a_1}$ $\frac{\partial \mathcal{L}}{\partial \theta_1} = \frac{\partial \mathcal{L}}{\partial a_1} \cdot \frac{\partial a_1}{\partial \theta_1}$



Dimension analysis

- \circ To make sure everything is done correctly \rightarrow "Dimension analysis"
- The dimensions of the gradient w.r.t. w^l must be equal to the dimensions of the respective weight w^l

$$\dim\left(\frac{\partial \mathcal{L}}{\partial a^l}\right) = \dim(a^l)$$

$$\dim\left(\frac{\partial \mathcal{L}}{\partial w^l}\right) = \dim(w^l)$$

Dimension analysis

• For
$$\frac{\partial \mathcal{L}}{\partial a^l} = \left(\frac{\partial a^{l+1}}{\partial x^{l+1}}\right)^T \frac{\partial \mathcal{L}}{\partial a^{l+1}}$$

$$\dim(a^{l}) = d_{l}$$
$$\dim(w^{l}) = d_{l-1} \times d_{l}$$

$$[d_l \times 1] = [d_{l+1} \times d_l]^T \cdot [d_{l+1} \times 1]$$

• For
$$\frac{\partial \mathcal{L}}{\partial w^l} = \frac{\partial a^l}{\partial w^l} \cdot \left(\frac{\partial \mathcal{L}}{\partial w^l}\right)^T$$

$$[d_{l-1} \times d_l] = [d_{l-1} \times 1] \cdot [1 \times d_l]$$

Dimensionality analysis: An Example

•
$$d_{l-1} = 15$$
 (15 neurons), $d_l = 10$ (10 neurons), $d_{l+1} = 5$ (5 neurons)
• Let's say $a^l = (w^l)^T x^l$



So, Backprop, what's the big deal?

- Backprop is as simple as it is complicated
- Mathematically, just the chain rule
 - Found some time around the 1700s by I. Newton and Leibniz, who invented calclulus
 - That simple, that we can even automate it
- However, why is it that we can train a highly non-complex machine with many local optima, like neural nets, with a strongly local learning algorithm like Backprop?
 - Why even is it a good choice?
 - Not really known, even today

Summary

UVA DEEP LEARNING COURSE EFSTRATIOS GAVVES

MODULAR LEARNING- PAGE 92

• Modularity in Neural Networks

Neural Network Modules

Neural Network Cheatsheet

• Backpropagation

Reading material

o Chapter 6

• Efficient Backprop, LeCun et al., 1998