CS584 Natural Language Processing

Deep feedforward networks

Department of Computer Science
Yue Ning
yue.ning@stevens.edu

Late Submission Policy

- **10%** penalty for late submission within **<u>24 hours</u>**.
- **40%** penalty for late submissions within **<u>24-48 hours</u>**.
- After 48 hours, you get **NO** points on the assignment.

Plan for today

- Deep Feedforward Networks
- Regularization for Deep Learning
- Optimization for Training Deep Models

Deep Feedforward Networks

- Also called feedforward neural networks, or multilayer perceptrons (MLPs)
- Goal: approximate some function f* (target function)
  - for a classifier, y=f*(x) maps an input **x** to a category **y**. It defines a mapping y=f(x;θ) and learns the parameters θ that result in the best function approximation.
- Information x flows through the function f, and finally to the output y.
- <u>No feedback connections</u> in which outputs of the model are fed back into itself.

Deep Feedforward Networks

- <u>w/o feedback connections</u>

(a) Recurrent neural network

(b) Forward neural network

Feedforward neural networks

- They are typically represented by composing together many different functions.
- Given three functions $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ connected in a chain, to form

$$f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$$

  - $f^{(1)}$ **first layer**
  - $f^{(2)}$ **second layer**
  - $f^{(3)}$ **output layer**
  - **depth** = 3

hidden layers

- Hidden layers
  - the training data does not show the desired output for each of these layers.

Example: Learning XOR

- XOR function y=f*(x)
  - When exactly one of these binary values is equal to 1, the XOR function returns 1. Otherwise, it returns 0.
- Data set:
  - $\mathbb{X} = \{[0,0]^\top, [0,1]^\top, [1,0]^\top, \text{ and } [1,1]^\top\}$
  - y = { [0], [1], [1], [0] }

A B	O
0 0	0
0 1	1
1 0	1
1 1	0

- Our model learns a function y=f(x;θ), and our learning algorithm will adapt the parameters θ to make f as similar as possible to f*.

Example: Learning XOR

- We can treat this problem as a <u>regression</u> problem and use a mean squared error (MSE) loss function.
- Evaluated on our whole training set, the MSE loss function:

$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{\boldsymbol{x} \in \mathbb{X}} \left(f^*(\boldsymbol{x}) - f(\boldsymbol{x}; \boldsymbol{\theta}) \right)^2$$

- Suppose that we choose a linear model

$$f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^\top \boldsymbol{w} + b$$

- We can minimize J(θ) in closed form with respect to w and b using the normal equations
  - w=0 and b=½
- The linear model simply outputs 0.5 everywhere. Why?

Example: Learning XOR

- We can treat this problem as a <u>regression</u> problem and use a mean squared error (MSE) loss function.
- Evaluated on our whole training set, the MSE loss function:

$$\boldsymbol{\theta}))^2$$

An analytical solution to the linear regression problem with a least-squares cost function.
see <u>least squares approximation</u>

- Su

$$b$$

- We can minimize J(θ) in closed form with respect to w and b using the normal equations
  - w=0 and b=½
- The linear model simply outputs 0.5 everywhere. Why?

Example: Learning XOR

- Cannot be separated with a single line

- Let's use a simple feedforward network

Example: Learning XOR

- Use a simple feedforward network with one hidden layer containing two hidden units.
- Use a nonlinear function as activation function in hidden units.

Example: Learning XOR

- The network now contains two functions chained together
  - $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}) = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c})$
  - $y = f^{(2)}(\mathbf{h}; \mathbf{w}, b) = \mathbf{h}^\top \mathbf{w} + b$
  - $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = f^{(2)}(f^{(1)}(\boldsymbol{x}))$
- In modern neural networks, the default recommendation is to use the rectified linear unit, or ReLU
  - g(z) = max{0, z}

- Our complete network is

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

Example: Learning XOR

- Our complete network is

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

- We can then specify a solution to the XOR

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

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

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

$$b = 0$$

Example: Learning XOR

- Our complete network is

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

- (1) Given input matrix X $\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$
- (2) multiply the input matrix by the first layer's weight matrix

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \qquad \boldsymbol{X}\boldsymbol{W} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$

- (3) add the bias vector c, to obtain $\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$

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

Example: Learning XOR

- Our complete network is

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

- (4) apply the rectified linear transformation, computing h

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

- (5) finish with multiplying by the weight vector w (b=0)

$$\boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \qquad \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \Leftrightarrow \boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Example: Learning XOR

- Our complete network is

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

- (4) apply the rectified linear transformation, computing h

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

plot this!

- (5) finish with multiplying by the weight vector w (b=0)

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

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \Leftrightarrow \boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Example: Learning XOR

- (Right) In the transformed space represented by the features extracted by a neural network, a linear model can now solve the problem.

Building a Machine Learning Algorithm

- Nearly all deep learning algorithms can be described as particular instances of a fairly simple recipe: combine
  - a specification of a dataset,
  - a cost function,
  - an optimization procedure
  - and a model.

- **Optimization**
  - For <u>linear</u> models, we could use normal equations, closed form optimization
  - For <u>nonlinear</u> models, it requires us to choose an iterative numerical optimization procedure, such as <u>gradient descent</u>.

Gradient-Based Learning

- The largest difference between the linear models and neural networks
  - the **nonlinearity** of a neural network causes most interesting loss functions to become **nonconvex**.

- Thus, neural networks are usually trained by using iterative, gradient-based optimizers that merely drive the cost function to a very low value

➔ Cost Functions and Output Units

Cost Functions

- Learning Conditional Distributions with Maximum Likelihood
- Negative log-likelihood, equivalently, cross-entropy between the training data and the model distribution

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} \log p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x})$$

The specific form of the cost function changes from model to model, depending on the specific form of $\log p_{\text{model}}$.

- if $p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; f(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{I})$, then we recover the mean squared error cost: proof: chapter 3.1.1 PRML

$$J(\theta) = \frac{1}{2}\mathbb{E}_{\mathbf{x},\mathbf{y}\sim\hat{p}_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x};\boldsymbol{\theta})||^2 + \text{const}$$

Cost Functions

- <u>Learning Conditional Statistics</u>
- Instead $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$, we want to learn just one conditional statistic of **y** given **x**
- Design the cost function to have its minimum lie on the function that maps **x** to the expected value of **y** given **x**
- To optimize the problem

$$f^* = \arg\min_f \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x})||^2$$

Different cost functions give different statistics.
Mean Absolute Error:

$$f^* = \arg\min_f \mathbb{E}_{\mathbf{x},\mathbf{y}\sim p_{\text{data}}} ||\boldsymbol{y} - f(\boldsymbol{x})||_1$$

Output Units

- **Linear** Units
  - Based on an affine transformation with no nonlinearity.
  - Given feature $\boldsymbol{h}$, a layer of linear output units produces a vector

$$\hat{\boldsymbol{y}} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$$

- An example problem
  - Suppose we seek to predict increment (either positive or negative) of a stock's price based on its prices in the past.

Output Units

- **Sigmoid** Units
  - Many tasks require predicting the value of a binary variable y.
  - Classification problems with two classes.

$$\hat{y} = \sigma\left(\boldsymbol{w}^\top \boldsymbol{h} + b\right)$$

- An example problem
  - Provide historical weather data and predict the <u>chance of rain</u> for the next day

Output Units

- **Softmax** Units
  - Represent a probability distribution over a discrete variable with *n* possible values
  - a generalization of the sigmoid function, which was used to represent a probability distribution over a binary variable.

$$\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$$

The softmax function can then exponentiate and normalize z to obtain the desired $\hat{y}$

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

- An example problem
  - Handwritten digit classification

Hidden Units

- An issue that is unique to feedforward neural nets
  - how to choose the <u>type</u> of hidden unit to use in the hidden layers of the model.
  - an extremely active research area and does not yet have many definitive guiding theoretical principles.

- Hidden units use the **activation functions**
  - they are typically used on top of an affine transformation:

$$\boldsymbol{h} = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b})$$

Activation Functions

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

tanh: $f(x) = 2\sigma(2x) - 1$

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

Leaky ReLU: $f(x) = \max(\alpha x, x)$

Maxout

$$\max(\mathbf{w}_1^T\mathbf{x} + b_1, \mathbf{w}_2^T\mathbf{x} + b_2)$$

ELU: $f(x) = \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$

*** good default choice**

Architecture Design

- **Architecture** refers to the overall structure of the network:
  - how many units it should have and how these units should be connected to each other
- Most neural networks
  - are organized into groups of units called layers
  - arrange these layers in a chain structure
  - each layer being a function of the layer that preceded it

$$\boldsymbol{h}^{(1)} = g^{(1)}\left(\boldsymbol{W}^{(1)\top}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)$$

$$\boldsymbol{h}^{(2)} = g^{(2)}\left(\boldsymbol{W}^{(2)\top}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right)$$

- main architectural considerations
  - choosing the depth and width of each layer

Architecture Design

Different neural networks

The depth and width of hidden layers are hyper-parameters!

Before Regularization and Optimization

- Basics of Convex Optimization
  - Convex Sets
  - Convex Functions
  - Convex Optimization

Convex Set

- **line segment**: between x_1 and x_2 : all points

  with $0 \leq \theta \leq 1$ $\quad x = \theta x_1 + (1-\theta)x_2$

- **convex set**: contains line segment between any two points in the set

$$x_1, x_2 \in C, \quad 0 \leq \theta \leq 1 \quad \Longrightarrow \quad \theta x_1 + (1-\theta)x_2 \in C$$

- **examples** (one convex, two nonconvex sets)

Convex Function

Definition 1. *A function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if its domain is a convex set and for all x, y in its domain, and all $\lambda \in [0, 1]$, we have*

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y).$$

Figure 1: An illustration of the definition of a convex function

Convex Function

- Examples on $\mathbb{R}$
  - Affine: $ax + b$ over $\mathbb{R}$ for any $a, b \in \mathbb{R}$
  - Exponential: e^{ax} over $\mathbb{R}$ for any $a \in \mathbb{R}$
  - Power: x^p over $(0, +\infty)$ for $p \geq 1$ or $p \leq 0$
  - Powers of absolute value: $|x|^p$ over $\mathbb{R}$ for $p \geq 1$
  - Negative entropy: $x \ln x$ over $(0, +\infty)$
- Examples on $\mathbb{R}^n$
  - Affine function $f(x) = a'x + b$ with $a \in \mathbb{R}^n$ and $b \in \mathbb{R}$
  - Euclidean, l_1, and l_∞ norms
  - General l_p norms

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p} \quad \text{for } p \geq 1$$

Convex Function

Important examples in Machine Learning

- SVM loss (hinge loss):

$$f(w) = \left[1 - y_i x_i^T w\right]_+$$

- Binary logistic loss:

$$f(w) = \log\left(1 + \exp(-y_i x_i^T w)\right)$$

Convex Optimization

Definition

- An optimization problem is **convex** if its objective is a convex function, the inequality constraints f_j are convex, and the equality constraints h_j are affine

$$
\begin{aligned}
\underset{x}{\text{minimize}} \quad & f_0(x) && \text{(Convex function)} \\
\text{s.t.} \quad & f_i(x) \leq 0 && \text{(Convex sets)} \\
& h_j(x) = 0 && \text{(Affine)}
\end{aligned}
$$

Convex Optimization

- Examples
  - Least squares regression: $\min_{\mathbf{w}} \left|\left|\mathbf{X}\mathbf{w} - \mathbf{y}\right|\right|_2^2$.
  - Logistic regression: $\min_{\mathbf{w}} \sum_j \log\left(1 + \exp\left(-y_j \mathbf{w}^T \mathbf{x}_j\right)\right)$.
  - SVM: $\min_{\mathbf{w},b} \left|\left|\mathbf{w}\right|\right|_2^2 + \lambda \sum_j \left[1 - y_j\left(\mathbf{w}^T \mathbf{x}_j + b\right)\right]_+$.
  - LASSO: $\min_{\mathbf{w}} \left|\left|\mathbf{X}\mathbf{w} - \mathbf{y}\right|\right|_2^2$; $s.t.\ \left|\left|\mathbf{w}\right|\right|_1 \leq t$.

Local and Global Optima

- Return local optimum is very common in optimization

- Property of convex optimization
  - Every local optimum is global optimum.

Regularization for Deep Learning

- A central problem in machine learning
  - Neural networks are prone to overfitting
  - How to make an algorithm that perform well on **new** inputs

- Developing more effective regularization strategies has been one of the major research efforts in the field.

- Several strategies for how to create such a large, deep regularized model.

Parameter Norm Penalties

- Limiting the capacity of models,such as neural networks, linear regression, or logistic regression, by adding a parameter norm penalty $\Omega(\boldsymbol{\theta})$ to the objective function *J.* the regularized objective function $\tilde{J}$

$$\tilde{J}(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta})$$

where $\alpha \in [0, \infty)$ is a hyperparameter that weights the relative contribution of the norm penalty term Ω relative to *J*.

α = 0: no regularization

large α: more regularization

△ leave the <u>biases</u> unregularized of the affine transformation at each layer. The biases require less data than the weights to fit accurately.

Vector Norms

- The ℓ_p norm: $\|\mathbf{x}\|_p := \left(\sum_i |x_i|^p\right)^{1/p}$.
- The ℓ_2 norm: $\|\mathbf{x}\|_2 = \left(\sum_i x_i^2\right)^{1/2}$ (the Euclidean norm).
- The ℓ_1 norm $\|\mathbf{x}\|_1 = \sum_i |x_i|$.
- The ℓ_∞ norm is defined by $\|\mathbf{x}\|_\infty = \max_i |x_i|$.

Matrix Norm

- Matrix norm corresponding to given vector norm defined by

$$\|A\| = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|}$$

- **Matrix Condition Number**
  - Condition number of square nonsingular matrix A

$$\kappa(A) = \|A\| \|A^{-1}\|$$

  - By convention, k(A) = ∞ if A singular
  - ill-conditioned matrix - large condition number
    - A small change in the input results in a surprisingly large change in the computed solution.

L2 Parameter Regularization

- Commonly known as **weight decay**, **ridge regression** or **Tikhonov regularization**.

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2$$

- We assume no bias parameter, so θ is just w. A model has the following total objective function

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2}\boldsymbol{w}^\top \boldsymbol{w} + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

L1 Parameter Regularization

- L1 regularization on the model parameter w is defined as

$$\Omega(\boldsymbol{\theta}) = ||\boldsymbol{w}||_1 = \sum_i |w_i|$$

as the sum of absolute values of the individual parameters.

- the regularized objective function is given by

$$\tilde{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha ||\boldsymbol{w}||_1 + J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$$

Dropout

- Dropout (Srivastava et al.,2014) provides a computationally inexpensive but powerful method of regularizing a broad family of models.
- At training (each iteration)
  - each unit is retained with a probability p. (randomly)
- At test (prediction)
  - the network is used as a whole.
  - the weights are scaled-down by a factor of p.

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

(a) At training time (b) At test time

Dropout

- Randomly dropping (setting to 0) half of the neurons in the network in each training example in the stochastic-gradient training.
- For example, a two hidden layer multi-layer perceptron:
  - $h^{[1]} = g(W^{[1]}x+b^{[1]})$
  - $h^{[2]} = g(W^{[2]}h^{[1]}+b^{[2]})$
  - $y = W^{[3]}h^{[2]}$
- Applying dropout:
  - $m^{[1]} \sim \text{Bernouli}\ (p^{[1]})$
  - $h'^{[1]} = m^{[1]} \odot h^{[1]}$
  - $h^{[2]} = g(W^{[2]}h'^{[1]}+b^{[2]})$
  - $m^{[2]} \sim \text{Bernouli}\ (p^{[2]})$
  - $h'^{[2]} = m^{[2]} \odot h^{[2]}$
  - $y = W^{[3]}h'^{[2]}$

Why Does Dropout Work?

- In training, dropout forces the network to make decision based on part of the features.

- Dropout is a regularization[1]
  - Alleviate overfitting.
  - Like the L1 and L2 norm regularizations.
  - But dropout is empirically better.

[1]. Wager, Wang, & Liang. Dropout Training as Adaptive Regularization. In NIPS, 2013.

Challenges in Neural Network Optimization

- Of all the many optimization problems involved in deep learning, the most difficult part is **neural network training**.

- When training neural networks, we must confront the general non convex case.

- Several of the most prominent challenges involved in optimization for training deep models.
  - Ill-Conditioning
  - Local Minima
  - Saddle Points

Ill-Conditioning

- The most prominent is ill-conditioning of the Hessian matrix

- **Hessian matrix** is a square matrix of second-order partial derivatives of a scalar-valued function.
  - Suppose f : ℝ^n → ℝ
    - input: a vector x ∈ ℝ^n
    - output: a scalar f(x) ∈ ℝ
  - If all second partial derivatives of f exist and are continuous over the domain of the function.

$$\mathbf{H}_{i,j} = \frac{\partial^2 f}{\partial x_i \partial x_j}$$

Ill-Conditioning

- The most prominent is ill-conditioning of the Hessian matrix

- **Hessian matrix**

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1\, \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1\, \partial x_n} \\ \frac{\partial^2 f}{\partial x_2\, \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2\, \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n\, \partial x_1} & \frac{\partial^2 f}{\partial x_n\, \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Ill-Conditioning

- Gradient descent often does not arrive at a critical point of any kind

- (Left) The solid curve: running average of all gradient norms. The gradient norm clearly increases over time, rather than decreasing as we would expect.
- (Right) Despite the increasing gradient, the training process is reasonably successful. The validation set classification error decreases to a low level.
- learning becomes very slow despite the presence of a strong gradient

more: https://www.deeplearningbook.org/contents/numerical.html
https://www.deeplearningbook.org/contents/optimization.html

Local Minima

- With nonconvex functions, such as neural nets, it is possible to have many local minima.
- Nearly any deep model is essentially guaranteed to have an extremely large number of local minima.

A test that can rule out local minima as the problem is plotting the norm of the gradient over time. If the norm of the gradient does not shrink to insignificant size, the problem is neither local minima nor any other kind of critical point.

Saddle Points

- In low-dimensional spaces
  - local minima are common
- In higher-dimensional spaces
  - local minima are rare, and saddle points are more common. (# saddle points > local minima/maxima)

Dauphin et al. (2014) introduced a saddle-free Newton method for second-order optimization and showed that it improves significantly over the traditional version. Second-order methods remain difficult to scale to large neural networks, but this saddle-free approach holds promise if it can be scaled.

[Identifying and attacking the saddle point problem in high-dimensional non-convex optimization]

Basic Algorithms

- **Gradient descent** algorithm that follows the gradient of an entire training set downhill.

- Accelerated Algorithms
  - Stochastic/Minibatch Gradient Descent
  - Momentum
  - Nesterov Momentum

An overview of gradient descent optimization algorithms: https://arxiv.org/pdf/1609.04747.pdf

Stochastic/Minibatch Gradient Descent

- The most used optimization algorithms
- To estimate the gradient by taking the average gradient on a **minibatch** of *m* examples drawn i.i.d from the data-generating distribution.
  - the learning rate is crucial.

Algorithm 8.1 Stochastic gradient descent (SGD) update

Require: Learning rate schedule $\epsilon_1, \epsilon_2, \ldots$
Require: Initial parameter $\boldsymbol{\theta}$
  $k \leftarrow 1$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \hat{\boldsymbol{g}}$
    $k \leftarrow k + 1$
  end while

Momentum

- SGD can sometimes be slow
- **Momentum** (Polyak, 1964) is designed to accelerate learning.
- **Momentum** derives from a physical analogy, in which the negative gradient is a force moving a particle through parameter space,according to Newton's laws of motion.

Momentum

- It accumulates an exponentially decaying moving average of past gradients and continues to move in their direction
  - velocity vector **v** may be regarded as the momentum of the particle
  - A hyperparameter α ∈[0,1) determines how quickly the contributions of previous gradients exponentially decay.

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}.$$

http://www.columbia.edu/~nq6/publications/momentum.pdf (1999)

Momentum

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.
  end while

Parameter Initialization Strategies

- Deep learning models are iterative and require the user to specify some **initial point** from which to begin the iterations.
- The initial point can determine whether the algorithm converges at all

1. initialize the weights of a fully connected layer with *m* inputs and *n* outputs by sampling each weight from

$$\mathrm{W}_{i,j} \sim U\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right)$$

2. normalized initialization [Glorot and Bengio(2010)]

$$\mathrm{W}_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

Algorithms with Adaptive Learning Rates

- The cost is often highly sensitive to some directions in parameter space
- Momentum algorithm can mitigate these issues, but it introduces another hyperparameter.

- Some algorithms
  - AdaGrad (Duchi et al.,2011)
  - RMSProp (Hinton, 2012)
  - Adam (Kingma and Ba, 2014)

Adaptive learning rates

- Popular and simple idea: reduce the learning rate by some factor every few epochs.
  - At the beginning, we are far from the destination, so we use larger learning rate
  - After several epochs, we are close to the destination, so we reduce the learning rate
  - E.g. 1/t decay: $\eta^t = \frac{\eta}{\sqrt{t+1}}$

- Learning rate cannot be one-size-fits-all
  - Giving different parameters different learning rates

Adaptive learning rates

- Divide the learning rate of each parameter by the root mean square of its previous derivatives.

$$\eta^t = \frac{\eta}{\sqrt{t+1}}, g^t = \frac{\partial L(w^t)}{\partial w}$$

- Vanilla gradient descent:

$$w^{t+1} \leftarrow w^t - \eta^t g^t$$

- Adagrad:

$$w^{t+1} \leftarrow w^t - \frac{\eta^t}{\sigma^t} g^t$$

σ^t : root mean square of the previous derivatives of parameter w

AdaGrad

$$
\begin{aligned}
w^{(1)} &\leftarrow w^{(0)} - \frac{\eta^{(0)}}{\sigma^{(0)}} g^{(0)} & \sigma^{(0)} &= \sqrt{(g^{(0)})^2 + \epsilon} \\
w^{(2)} &\leftarrow w^{(1)} - \frac{\eta^{(1)}}{\sigma^{(1)}} g^{(1)} & \sigma^{(1)} &= \sqrt{\tfrac{1}{2}[(g^{(0)})^2 + (g^{(1)})^2] + \epsilon} \\
w^{(3)} &\leftarrow w^{(2)} - \frac{\eta^{(2)}}{\sigma^{(2)}} g^{(2)} & \sigma^{(2)} &= \sqrt{\tfrac{1}{3}[(g^{(0)})^2 + (g^{(1)})^2 + (g^{(2)})^2] + \epsilon} \\
&\ldots & &\ldots \\
w^{(t+1)} &\leftarrow w^{(t)} - \frac{\eta^{(t)}}{\sigma^{(t)}} g^{(t)} & \sigma^{(t)} &= \sqrt{\tfrac{1}{t+1} \textstyle\sum_{i=1}^{t} (g^{(i)})^2 + \epsilon}
\end{aligned}
$$

ε is a smoothing term that avoids division by zero (usually on the order of 1e − 8)

AdaGrad

- Divide the learning rate of each parameter by the root mean square of its previous derivatives

$$w^{t+1} \leftarrow w^t - \frac{\eta^t}{\sigma^t} g^t$$

where $\eta^t = \frac{\eta}{\sqrt{t+1}}$ and $\sigma^{(t)} = \sqrt{\frac{1}{t+1}\sum_{i=1}^{t}(g^{(i)})^2 + \epsilon}$.

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\sqrt{\sum_{i=0}^{t}(g^i)^2}} g^t$$

Contradiction?

- Vanilla gradient descent:

$$w^{t+1} \leftarrow w^t - \eta^t g^t$$

Larger gradient, larger step

- Adagrad:

$$w^{t+1} \leftarrow w^t - \frac{\eta}{\underbrace{\sqrt{\sum_{i=0}^{t} (g^i)^2}}_{\text{larger gradient smaller step}}} \underbrace{g^t}_{\text{larger gradient larger step}}$$

Intuitive Reason

- Adagrad aims at capturing the relative changes compared to previous gradients (How surprise it is)

g^0	g^1	g^2	g^3	g^4	...
0.001	0.001	0.003	0.002	0.1	...
g^0	g^1	g^2	g^3	g^4	...
10.8	20.9	31.7	12.1	0.1	...

Larger gradient, larger steps?

Comparison between different parameters

Second Derivative

Note that $\frac{\partial^2 y}{\partial x^2} = 2a$. The best step is $\frac{|\text{first derivative}|}{\text{second derivative}}$

Comparison between different parameters

Second Derivative

- Use first derivative to estimate second derivative

Sampling multiple points on the first-derivative curves

Optimization Strategies

- Feature scaling
- Batch normalization
- Supervised pretraining

Feature Scaling

- Assume the samples $x_i, \ldots, x_n$ are one-dimensional.
- **Min-max normalization**

$$x_i' = \frac{x_i - \min(x_i)}{\max(x_i) - \min(x_i)}$$

After the scaling, the samples $x_1', \cdots, x_n'$ are in [0,1]

- **Standardization** $$x_i' = \frac{x_i - \hat{\mu}}{\hat{\sigma}}$$

  - $\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i$ is the sample mean
  - $\hat{\sigma}^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \hat{\mu})^2$ is the sample variance

After the scaling, the samples $x_1', \cdots, x_n'$ have zero mean and unit variance

Why Feature Scaling

- Make the scales of all the features comparable.
- Faster convergence

Batch Normalization

- Batch normalization (Ioffe and Szegedy, 2015) is one of the most exciting recent innovations in optimizing deep neural networks.

- Feature standardization of hidden layers

Batch Normalization

- Let $\mathbf{x}^{(k)} \in \mathbb{R}^d$ be the output of k-th hidden layer
- $\hat{\boldsymbol{\mu}} \in \mathbb{R}^d$ sample mean of $\mathbf{x}^{(k)}$ evaluated on a batch of samples
- $\hat{\boldsymbol{\sigma}} \in \mathbb{R}^d$ sample std of $\mathbf{x}^{(k)}$ evaluated on a batch of samples

- Standardization $z_j^{(k)} = \frac{x_j^{(k)} - \hat{\mu}_j}{\hat{\sigma}_j + 0.001}$, for $j = 1, \cdots, d$.

Batch Normalization

- Let $\mathbf{x}^{(k)} \in \mathbb{R}^d$ be the output of k-th hidden layer
- $\hat{\boldsymbol{\mu}} \in \mathbb{R}^d$ sample mean of $\mathbf{x}^{(k)}$ evaluated on a batch of samples
- $\hat{\boldsymbol{\sigma}} \in \mathbb{R}^d$ sample std of $\mathbf{x}^{(k)}$ evaluated on a batch of samples
- $\boldsymbol{\gamma} \in \mathbb{R}^d$ scaling parameter (**trainable**)
- $\boldsymbol{\beta} \in \mathbb{R}^d$ shifting parameter (**trainable**)

- Standardization $z_j^{(k)} = \frac{x_j^{(k)} - \hat{\mu}_j}{\hat{\sigma}_j + 0.001}$, for $j = 1, \cdots, d$.
- Scale and shift $x_j^{(k+1)} = z_j^{(k)} \circ \gamma_j + \beta_j$, for $j = 1, \cdots, d$.

- Use backpropagation to update γ_j and β_j

Why Batch Normalization

- Faster convergence
- Why it works?
  - Weights change values after each backprop so will the input to each layer.
  - Batch norm is a way to make the range of inputs to each layer more consistent and thus making it easier for optimize the weights in that layer.

paper: How Does Batch Normalization Help Optimization?

Supervised Pretraining

- Sometimes, directly training a model to solve a specific task can be too ambitious if
  - the model is complex and hard to optimize
  - or the task is very difficult
- It is more effective to train a simpler model to solve the task, then make the model more complex.
- **Pretraining**
  - greedy supervised pretraining

Summary

- Deep Feedforward Networks
  - Gradient-based learning
    - Cost functions
    - Output Units
  - Hidden units
- Basics of convex optimization
- Regularization for Deep Learning
  - Parameter norm penalties (l1,l2 regularization)
  - Dropout
- Optimization for Training Deep Models
  - Challenges in neural optimization
  - Accelerated algorithms
  - Algorithms with adaptive learning rates
  - Batch normalization

Practice: Sentiment analysis

- Data Set: labeled reviews from Yelp

- # samples 1000
- # positive 500
- # negative 500

	sentence	label
0	Wow... Loved this place.	1
1	Crust is not good.	0
2	Not tasty and the texture was just nasty.	0
3	Stopped by during the late May bank holiday of...	1
4	The selection on the menu was great and so wer...	1
5	Now I am getting angry and I want my damn pho.	0
6	Honeslty it didn't taste THAT fresh.)	0
7	The potatoes were like rubber and you could te...	0
8	The fries were great too.	1
9	A great touch.	1
10	Service was very prompt.	1
11	Would not go back.	0
12	The cashier had no care what so ever on what I...	0
13	I tried the Cape Cod ravoli, chicken,with cran...	1
14	I was disgusted because I was pretty sure that...	0

Practice: Sentiment analysis

- Word count as feature - Bag-of-words (BOW)

```
sentences = ['Would not go back', 'Crust is not good']
```

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(min_df=0, lowercase=False)
vectorizer.fit(sentences)
vectorizer.vocabulary_
```

```
{'Would': 1, 'not': 6, 'go': 3, 'back': 2, 'Crust': 0, 'is': 5, 'good': 4}
```

```
vectorizer.transform(sentences).toarray()
```

```
array([[0, 1, 1, 1, 0, 0, 1],
       [1, 0, 0, 0, 1, 1, 1]])
```

Practice: Sentiment analysis

Load data

```
import pandas as pd
filepath ='data/sentiment_analysis/yelp_labelled.txt'
df = pd.read_csv(filepath, names=['sentence', 'label'], sep='\t')
print(df.iloc[0])
```

```
sentence    Wow... Loved this place.
label                               1
Name: 0, dtype: object
```

Split data

```
from sklearn.model_selection import train_test_split

sentences = df['sentence'].values
y = df['label'].values
sentences_train, sentences_test, y_train, y_test = train_test_split(
   sentences, y, test_size=0.25, random_state=1000)
```

Vectorizer

```
from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer()
vectorizer.fit(sentences_train)

X_train = vectorizer.transform(sentences_train)
X_test  = vectorizer.transform(sentences_test)
X_train
```

```
<750x1714 sparse matrix of type '<class 'numpy.int64'>'
        with 7368 stored elements in Compressed Sparse Row format>
```

Practice: Sentiment analysis

Deep feedforward model

```
from keras.models import Sequential
from keras import layers

input_dim = X_train.shape[1]  # Number of features
print('input_dim',input_dim)

model = Sequential()
model.add(layers.Dense(10, input_dim=input_dim, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(10, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
model.summary()
```

```
input_dim 1714
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_75 (Dense)             (None, 10)                17150
_________________________________________________________________
dropout_28 (Dropout)         (None, 10)                0
_________________________________________________________________
dense_76 (Dense)             (None, 10)                110
_________________________________________________________________
dense_77 (Dense)             (None, 1)                 11
=================================================================
Total params: 17,271
Trainable params: 17,271
Non-trainable params: 0
```

Practice: Sentiment analysis

Training

```
history = model.fit(X_train, y_train,
                    epochs=50,
                    verbose=False,
                    validation_data=(X_test, y_test),
                    batch_size=10)
```

```
loss, accuracy = model.evaluate(X_train, y_train, verbose=False)
print("Training Accuracy: {:.4f}".format(accuracy))
loss, accuracy = model.evaluate(X_test, y_test, verbose=False)
print("Testing Accuracy:  {:.4f}".format(accuracy))
```

```
Training Accuracy: 1.0000
Testing Accuracy:  0.8160
```

Part of speech (POS) tagging

Data

The [Penn Treebank]() is an annotated corpus of POS tags.

```python
import numpy as np
import nltk
from nltk.corpus import treebank
# nltk.download('treebank')
# nltk.download('universal_tagset')
sentences = treebank.tagged_sents(tagset='universal')
```

```python
print(sentences[0]) # a list of tuples (term, tag)
```

```
[('Pierre', 'NOUN'), ('Vinken', 'NOUN'), (',', '.'), ('61', 'NUM'), ('years', 'NOUN'), ('old', 'ADJ'), (',', '.'),
('will', 'VERB'), ('join', 'VERB'), ('the', 'DET'), ('board', 'NOUN'), ('as', 'ADP'), ('a', 'DET'), ('nonexecutive',
'ADJ'), ('director', 'NOUN'), ('Nov.', 'NOUN'), ('29', 'NUM'), ('.', '.')]
```

```python
print('#sentences: %s' % (len(sentences)))
tags = set([tag for sentence in treebank.tagged_sents() for _, tag in sentence])
print('#tags: %s  %s' % (len(tags), tags))
```

```
#sentences: 3914
#tags: 46  {"''", 'DT', 'VBN', 'EX', 'RBS', '$', ',', 'VB', 'POS', 'TO', 'SYM', 'VBD', 'RBR', 'NNPS', 'PDT', '-RRB-',
'CC', 'VBZ', 'WP', 'FW', 'JJ', 'NNS', 'IN', '``', ':', '-NONE-', 'CD', 'UH', 'VBG', '-LRB-', '#', 'WRB', 'NNP', '.',
'LS', 'MD', 'RP', 'RB', 'PRP$', 'JJS', 'JJR', 'VBP', 'NN', 'WP$', 'PRP', 'WDT'}
```

Data preprocessing

- split the tagged sentences into 3 datasets
  - training
  - validation
  - test

```
training_sentences = sentences[: int(.8 * len(sentences)) ]
testing_sentences = sentences[int(.8 * len(sentences)) :]

training_sentences = training_sentences[:int(.75 * len(training_sentences))]
validation_sentences = training_sentences[int(.75 * len(training_sentences)):]

```

Feature engineering

map the list of sentences to a list of dict features.

```python
def add_basic_features(sentence_terms, index):
    term = sentence_terms[index]
    return {
        'nb_terms': len(sentence_terms),
        'term': term,
        'is_first': index == 0,
        'is_last': index == len(sentence_terms) - 1,
        'is_capitalized': term[0].upper() == term[0],
        'is_all_caps': term.upper() == term,
        'is_all_lower': term.lower() == term,
        'prefix-1': term[0],
        'prefix-2': term[:2],
        'prefix-3': term[:3],
        'suffix-1': term[-1],
        'suffix-2': term[-2:],
        'suffix-3': term[-3:],
        'prev_word': '' if index == 0 else sentence_terms[index - 1],
        'next_word': '' if index == len(sentence_terms) - 1
                        else sentence_terms[index + 1]
    }

def untag(tagged_sentence):
    return [w for w, _ in tagged_sentence]

def transform_to_dataset(tagged_sentences):
    X, y = [], []
    for pos_tags in tagged_sentences:
        for index, (term, class_) in enumerate(pos_tags):
            # Add basic NLP features for each sentence term
            X.append(add_basic_features(untag(pos_tags), index))
            y.append(class_)
    return X, y
```

Feature engineering

For training, validation and testing sentences, we split the attributes into X (input variables) and y (output variables).

```
X_train, y_train = transform_to_dataset(training_sentences)
X_test, y_test = transform_to_dataset(testing_sentences)
X_val, y_val = transform_to_dataset(validation_sentences)
```

```
len(X_train),len(X_val),len(X_test)
```

```
(61014, 16016, 20039)
```

Feature engineering

example

```
1 print(sentences[0])
```

```
[('Pierre', 'NOUN'), ('Vinken', 'NOUN'), (',', '.'), ('61', 'NUM'), ('years',
'NOUN'), ('old', 'ADJ'), (',', '.'), ('will', 'VERB'), ('join', 'VERB'), ('th
e', 'DET'), ('board', 'NOUN'), ('as', 'ADP'), ('a', 'DET'), ('nonexecutive',
'ADJ'), ('director', 'NOUN'), ('Nov.', 'NOUN'), ('29', 'NUM'), ('.', '.')]
```

```
1 X_train[0]
```

```
{'nb_terms': 18,
 'term': 'Pierre',
 'is_first': True,
 'is_last': False,
 'is_capitalized': True,
 'is_all_caps': False,
 'is_all_lower': False,
 'prefix-1': 'P',
 'prefix-2': 'Pi',
 'prefix-3': 'Pie',
 'suffix-1': 'e',
 'suffix-2': 're',
 'suffix-3': 'rre',
 'prev_word': '',
 'next_word': 'Vinken'}
```

```
1 y_train[0]
```

```
'NOUN'
```

Features encoding

- convert our dict features to vectors

```
from sklearn.feature_extraction import DictVectorizer
# Fit our DictVectorizer with our set of features
dict_vectorizer = DictVectorizer(sparse=False)
dict_vectorizer.fit(X_train + X_test + X_val)
# Convert dict features to vectors
X_train = dict_vectorizer.transform(X_train)
X_test = dict_vectorizer.transform(X_test)
X_val = dict_vectorizer.transform(X_val)
```

- encode y as integers

```
from sklearn.preprocessing import LabelEncoder
# Fit LabelEncoder with our list of classes
label_encoder = LabelEncoder()
label_encoder.fit(y_train + y_test + y_val)
# Encode class values as integers
y_train = label_encoder.transform(y_train)
y_test = label_encoder.transform(y_test)
y_val = label_encoder.transform(y_val)
```

- convert integers (y) to one-hot encoding

```
# Convert integers to dummy variables (one hot encoded)
from keras.utils import np_utils
y_train = np_utils.to_categorical(y_train)
y_test = np_utils.to_categorical(y_test)
y_val = np_utils.to_categorical(y_val)
```

```
X_train.shape
```

```
(61014, 39684)
```

Building a DNN model using Keras

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation

model = Sequential([
    Dense(512, input_dim=X_train.shape[1]),
    Activation('relu'),
    Dropout(0.5),
    Dense(512),
    Activation('relu'),
    Dropout(0.5),
    Dense(256),
    Activation('relu'),
    Dense(y_train.shape[1], activation='softmax')
])
model.compile(loss='categorical_crossentropy',
              optimizer='adam', metrics=['accuracy'])
print(model.summary())
```

Building a DNN model using Keras

Model summary

Layer (type)	Output Shape	Param #
dense_12 (Dense)	(None, 512)	22647296
activation_9 (Activation)	(None, 512)	0
dropout_7 (Dropout)	(None, 512)	0
dense_13 (Dense)	(None, 512)	262656
activation_10 (Activation)	(None, 512)	0
dropout_8 (Dropout)	(None, 512)	0
dense_14 (Dense)	(None, 256)	131328
activation_11 (Activation)	(None, 256)	0
dense_15 (Dense)	(None, 12)	3084

```
Total params: 23,044,364
Trainable params: 23,044,364
Non-trainable params: 0
```

Training

```
hist = model.fit(X_train, y_train, epochs=5, batch_size=128,verbose=1,
                 shuffle=True, validation_data=(X_val, y_val))

```

```
Train on 61107 samples, validate on 19530 samples
Epoch 1/5
61107/61107 [==============================] - 120s 2ms/step - loss: 0.4426 - acc: 0.8641 - val_loss:
0.1642 - val_acc: 0.9499
Epoch 2/5
61107/61107 [==============================] - 117s 2ms/step - loss: 0.1250 - acc: 0.9611 - val_loss:
0.1457 - val_acc: 0.9548
Epoch 3/5
61107/61107 [==============================] - 116s 2ms/step - loss: 0.0835 - acc: 0.9743 - val_loss:
0.1449 - val_acc: 0.9577
Epoch 4/5
61107/61107 [==============================] - 115s 2ms/step - loss: 0.0694 - acc: 0.9783 - val_loss:
0.1445 - val_acc: 0.9581
Epoch 5/5
61107/61107 [==============================] - 115s 2ms/step - loss: 0.0525 - acc: 0.9841 - val_loss:
0.1390 - val_acc: 0.9593
```

Results

```
score = model.evaluate(X_test, y_test)
print(score)
```

```
20039/20039 [==============================] - 4s 178us/step
[0.1017228750231013, 0.9656170467588203]
```

reference: https://becominghuman.ai/part-of-speech-tagging-tutorial-with-the-keras-deep-learning-library-d7f93fa05537

STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

stevens.edu

Thank You