BZAN 554:
Deep Learning for Business Applications

Fundamentals of Neural Networks and
Getting Started with TensorFlow

Michel Ballings

Outline I

- A note on the technology used in this class: TensorFlow (TF)
- Start of field of Neural Networks: the Perceptron
- Continuous response variables
  - Architecture for a single continuous response
  - Architecture for multiple continuous responses
  - Loss functions for (multiple) continuous response
- Categorical response variables
  - Architecture for binary response
  - Output activation function for binary response
  - Loss function for binary response
  - Architecture for multi-label response
  - Output activation function for multi-label response
  - Loss function for multi-label response
  - Architecture for multi-class response
  - Output activation function for multi-class response
  - Loss function for multi-class response
  - Review Multi-label vs Multi-class
- Activation functions
  - What is an activation function, and do we need it?
  - What does a good activation function look like?
  - Sigmoid
  - Tanh

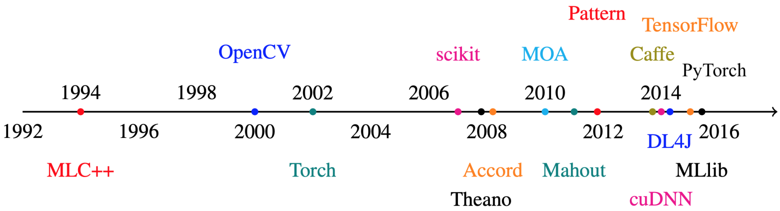
Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

History of Deep Learning Libraries



Why should I use TensorFlow?

- Open source with comprehensive ecosystem of tools, libraries and community resources
- Since version 2.0, much easier to develop models with eager execution
- Lower-level TensorFlow code allows for powerful models
- Robust and fast for production

TensorFlow APIs

TensorFlow has APIs available in several languages: Python (most complete), JavaScript, C++, Java, Go, and Swift (early release). Third party packages are available for Julia, R, and more.

In this course we use the Python API. See here for the latest information on API support: https://www.tensorflow.org/api_docs/

What is TensorFlow?

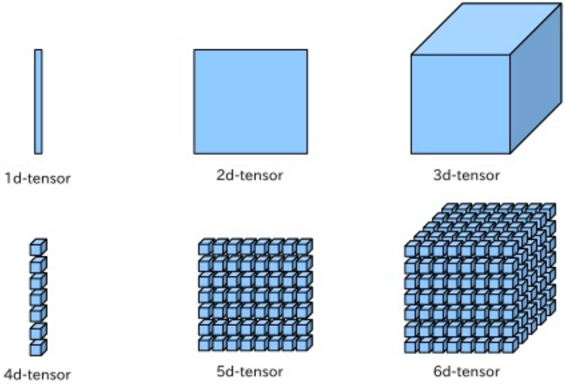
- Second generation open- source version (generation 1 was released on Feb 11, 2017) of Google's proprietary machine learning system DistBelief (started in 2011), created by the Google Brain team.
- TensorFlow can train and run deep neural networks for handwritten digit classification, image recognition, word embeddings, recurrent neural networks, sequence-to-sequence models for machine translation, natural language processing, and PDE (partial differential equation) based simulations.

What is TensorFlow?

- TensorFlow = Data Flow = Data Stream
- Tensor = multidimensional data array (see next slide)
- Flow = move the data through a directed graph of operations
- Dataflow programming defines a program as a series of connections emphasizing movement of data and looks like a graph. It is easy to parallelize and distribute computations. Hence it is the perfect software model for Neural Networks, which are also graphs. More information: https://en.wikipedia.org/wiki/Dataflow_programming and https://www.tensorflow.org/guide/graphs

Tensor?

Tensor is a general name for a multidimensional or an N-way array. For example, a 1-d tensor is a vector, a 2-d tensor is a matrix, a 3-d tensor is a cube. We can imagine that a 4-d tensor is a vector of cubes. In a similar way, a 5-d tensor is a matrix of cubes and a 6-d tensor is a cube of cubes.



Flow?

Think of the flowing data as water flowing through a pipe. If the pipes are part of a water treatment facility then the pipes will lead to a basin for treating the water, then flow through some more pipes into another basin for another treatment, and so on. The treatment in a basin is a metaphor for the operations (calculations).

An **operation** or **op** is a node in a TensorFlow Graph that takes one or more Tensor objects as input, and produces one or more Tensor objects as output.

TensorFlow uses a dataflow graph to represent your computation in terms of the dependencies between individual operations.

In this course...

...you can expect to see a TensorFlow implementation after the introduction of every new concept.

Outline I

- A note on the technology used in this class: TensorFlow (TF)
- Start of field of Neural Networks: the Perceptron
- Continuous response variables
  - Architecture for a single continuous response
  - Architecture for multiple continuous responses
  - Loss functions for (multiple) continuous response
- Categorical response variables
  - Architecture for binary response
  - Output activation function for binary response
  - Loss function for binary response
  - Architecture for multi-label response
  - Output activation function for multi-label response
  - Loss function for multi-label response
  - Architecture for multi-class response
  - Output activation function for multi-class response
  - Loss function for multi-class response
  - Review Multi-label vs Multi-class
- Activation functions
  - What is an activation function, and do we need it?
  - What does a good activation function look like?
  - Sigmoid
  - Tanh

Outline II

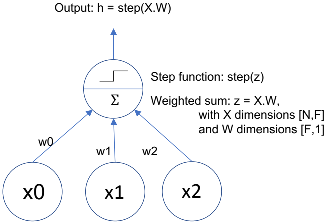
- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Start of field of Neural Networks: the Perceptron

A perceptron[1] is the simplest of neural network architectures. It has no hidden layers. It takes inputs, computes the weighted sum, applies an activation function (squashing function) and outputs the result.



Note: when all the neurons in a layer are connected to all the neurons in the previous layer then that is called a **dense layer**, or a **fully-connected layer**.

[1] F. Rosenblatt (1957). The perceptron, a perceiving and recognizing automaton. Cornell Aeronautical Laboratory

Perceptron

A perceptron computes a weighted sum of inputs:

$$z = w_0x_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n = \sum_{j=1}^{m} x_jw_j + b = \mathbf{X} \cdot \mathbf{W} \tag{1}$$

with X a matrix of dimensions [N,F] (N: instances, F: features), W a matrix of dimensions [F,1], w_0 the bias and x_0 the constant[2]. If we are training on a single record at a time then N=1.

Then, we apply a step function to that sum and output the result:

$$h_w(\mathbf{x}) = step(z) = step(\mathbf{X} \cdot \mathbf{W}) \tag{2}$$

[2] Note that I am including the constant and bias in respectively X and W for notational convenience. We can keep the constant separate, and also the bias, and then the equation would be $\mathbf{X} \cdot \mathbf{W} + \mathbf{b}$ as we had before.

Perceptron

The most common step functions in a perceptron are the *Heaviside step function* defined as

$$\hat{y} = o = heaviside(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ 1 & \text{if } z > 0, \end{cases} \tag{3}$$

and the sign function:

$$\hat{y} = o = sign(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ 1 & \text{if } z > 0. \end{cases} \tag{4}$$

Perceptron Learning Rule

The perceptron's training regime as proposed by Rosenblatt was inspired by Hebb's rule (1949). Hebb's idea was later summarized as 'Cells that fire together, wire together' by Lowel; the weight between two neurons increases when they fire at the same time.

Rosenblatt (1957) modified this rule to take into account the prediction error: connections that help reduce the error are reinforced. This is equivalent to stochastic gradient descent. The perceptron is given one training instance at a time, and for each instance it makes its predictions. The per-instance rule is shown here:

$$w_j = w_j + \eta\,(y - \hat{y})\,x_j \tag{5}$$

where

- w_j is the connection weight between input neuron j and the output neuron
- $\hat{y}$ is the output neuron's value for the current training instance.
- y is the target output for the current training instance.
- η is the learning rate.
- x_j is input variable j

Illustration of Perceptron Learning Rule

1. Let $\eta = 0.1$ and consider the following training data X and y

x_0	x_1	x_2	y
1	0	1	-1
1	-1	-1	1
1	-0.5	-1	1

2. Randomly initialize the weights in the perceptron: $w = \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix}$

3. Compute z and $\hat{y}$ for record [0] as follows:

$$z = X^{[0]}.W = (1, 0, 1). \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix} = 1 \tag{6}$$

$$\hat{y} = sign(z) = 1 \tag{7}$$

Illustration of Perceptron Learning Rule

4. Collect $\hat{y} = 1$, $y = -1$ for record [0], and $eta = 0.1$ and plug them into the update rule:

$$\begin{aligned}
w_0 &= w_0 + 0.1(-1 - 1)x_0 \\
&= w_0 - (0.2)1 \\
&= 1 - 0.2 \\
&= 0.8 \\
w_1 &= w_1 + 0.1(-1 - 1)x_1 \\
&= w_1 - (0.2)0 \\
&= -1 - 0 \\
&= -1 \\
w_2 &= w_2 + 0.1(-1 - 1)x_2 \\
&= w_2 - (0.2)1 \\
&= 0 - 0.2 \\
&= -0.2
\end{aligned} \tag{8}$$

Illustration of Perceptron Learning Rule

5. Repeat step 3 and 4 for the other two records and we're done. The **first important insight** is that if ($y = 1$ and $\hat{y} = 1$) or ($y = -1$ and $\hat{y} = -1$) then

$$\begin{aligned} w &= w + 0.1\,(0)\,x \\ &= w \end{aligned} \tag{9}$$

meaning it only learns when the prediction is wrong. The **second important insight** is that if the desired outcome (y) is negative (-1) and our prediction is wrong (+1) then the weight will be adjusted downwards (towards negative),

$$w = w + 0.1\,(-1-(1))\,x = w + 0.1\,(-2)\,x \tag{10}$$

and if the desired outcome (y) is positive (1) and our prediction is wrong (-1) then the weight will be adjusted upwards (towards positive)

$$w = w + 0.1\,(1-(-1))\,x = w + 0.1\,(2)\,x \tag{11}$$

Perceptrons fell into disfavor in 1969

- In 1969 Minsky and Papert pointed out that the perceptron has a major drawback: the perceptron learning rule converges if the two classes can be separated by a linear hyperplane, but problems arise for nonlinear problems.
- Because of this disappointment, researchers dropped perceptrons until the 1980s, when it was discovered that stacked perceptrons - called **Multi-Layer Perceptrons** can learn non-linear problems. Around that time other non-linear activation functions were introduced.

Perceptrons fell into favor in 1986

- In 1986 Rumelhart, Hinton and Williams published a breakthrough called **backpropagation** which consists in:
  - a forward pass (making a prediction) and computing the error
  - a backward pass through each layer to measure the contribution of each weight to the error and updating the weights to reduce the error
- Key contribution: change the step function to a sigmoid function. The step function has only flat segments and so the gradient is 0 everywhere, while the sigmoid has a non-zero derivative everywhere. This allows the gradient descent algorithm to make at least some progress in every weight update step.

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables

- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables

- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions

- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Outline I

- A note on the technology used in this class: TensorFlow (TF)
- Start of field of Neural Networks: the Perceptron
- Continuous response variables
    - Architecture for a single continuous response
    - Architecture for multiple continuous responses
    - Loss functions for (multiple) continuous response
- Categorical response variables
    - Architecture for binary response
    - Output activation function for binary response
    - Loss function for binary response
    - Architecture for multi-label response
    - Output activation function for multi-label response
    - Loss function for multi-label response
    - Architecture for multi-class response
    - Output activation function for multi-class response
    - Loss function for multi-class response
    - Review Multi-label vs Multi-class
- Activation functions
    - What is an activation function, and do we need it?
    - What does a good activation function look like?
    - Sigmoid
    - Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Architecture for a single continuous response

- Hidden nodes/layers: use sigmoid activation for now, we will see a large variety of activation functions in a subsequent session. Any number of hidden layers and neurons
- No activation is used in the output unless we want the output to be within a specific range. For example: we may want the output to be greater than 0 (use softplus), or within $[0, 1]$ (use sigmoid) or $[-1, 1]$ (use thanh). We will see the details of these activation functions in subsequent sessions and they are mentioned here for completeness.
- Loss: Typically the mean squared error (mse) is used. If there are a lot of outliers, we can use the mean absolute error (mae), or the Huber loss, which is a combination of both. The Huber loss is quadratic when the error $abs(y - \hat{y})$ is smaller than a threshold (say 3) and linear when the error is larger than the threshold. The quadratic part allows for faster (when difference > 1) and smoother convergence (when difference < 1), and the linear part reduces sensitivity to outliers. We will see the details of these losses in subsequent sessions and they are mentioned here for completeness

Architecture for continuous response

- Note: There are three TF APIs: Sequential, Functional, Subclassing. In this class we focus on the functional API.
- You may see the term Keras. Keras is a reference developed by F. Chollet in 2015 for implementing deep learning libraries. TensorFlow has implemented this reference (and added some features) and it is called **tf.keras**. The Keras reference is documented on `https://keras.io/` and the TensorFlow implementation is documented on `https://www.tensorflow.org/guide/keras/functional`
- See **RegressionTF.py**

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables

- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables

- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions

- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Architecture for multiple continuous response

- Everything is exactly the same as for a single response
- Average of losses is used to update the weights
- See **MultiOutputRegressionTF.py**

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables

- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables

- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions

- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Mean Squared Error

The Mean Squared Error (MSE) is the dominant choice for continuous responses. It is defined as follows for a model h_θ parameterized by the vector θ (θ is another popular way of denoting w). The partial derivative for parameter j is also provided.

$$\begin{aligned} MSE(y, h_\theta(X)) &= \frac{1}{n}\sum_{i=1}^{n}(h_\theta(X_i) - y_i)^2 \\ \frac{\partial MSE}{\partial \theta_j} &= \frac{2}{n}\sum_{i=1}^{n}(h_\theta(X_i) - y_i)\frac{\partial h_\theta(X_i)}{\partial \theta_j} \\ &= \frac{2}{n}\sum_{i=1}^{n}(\hat{y} - y_i)\frac{\partial \hat{y}}{\partial \theta_j} \end{aligned} \tag{12}$$

Specifically, for the weight vector θ connecting the last hidden layer and the output layer:

$$\begin{aligned} MSE(y, h_\theta(X)) &= \frac{1}{n}\sum_{i=1}^{n}(X_i\theta - y_i)^2 \\ \frac{\partial MSE}{\partial \theta_j} &= \frac{2}{n}\sum_{i=1}^{n}(X_i\theta - y_i)X_{i,j} \end{aligned} \tag{13}$$

Sometimes MSE is defined as 1/2 MSE, for mathematical convenience so that the derivative does not have 2 in the numerator. It can be accounted for in the learning rate.

Mean Squared Error

Advantages:

- Fast learning because predictions farther away from the true point result in a disproportionately larger loss. For example, being 5 points off from the true point results in a loss of 25, whereas being 10 points off (i.e., 2 times 5) results in a loss of 100 (4 times 25). A larger loss results in a larger gradient and a larger update to the weights.
- Easy to converge to minimum because as we get closer to the minimum the gradient becomes increasingly small (as opposed to constant); no bouncing around the minimum.

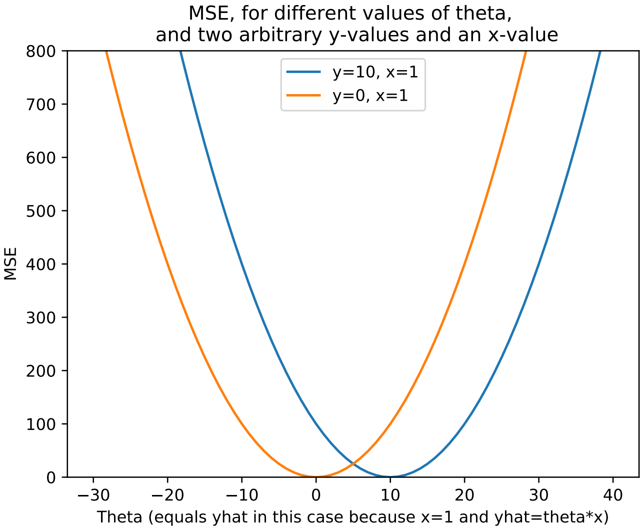
Disadvantages:

- Sensitive to outliers because differences are squared

Mean Squared Error

```python
#note: assume n = 1
import matplotlib.pyplot as plt
import numpy as np
#loss plot for two artibrarily chosen y-values and an x-value
#and different values of theta
theta = np.arange(-30,41)
x = 1
yhat = theta * x

y = 10 #true value is arbitrarily chosen to be 10
mse = (y - yhat)**2
plt.plot(yhat,mse, label = 'y=10, x=1')

y = 0 #true value is arbitrarily chosen to be 0
mse = (y - yhat)**2
plt.plot(yhat,mse,label = 'y=0, x=1')

plt.legend(loc="upper center")
plt.ylim(0, 800)
plt.title('MSE, for different values of theta, \n \
and two arbitrary y-values and an x-value')
plt.xlabel('Theta (equals yhat in this case because x=1 and \n \
yhat=theta*x)')
plt.ylabel('MSE')
plt.savefig('mse.pdf')
```

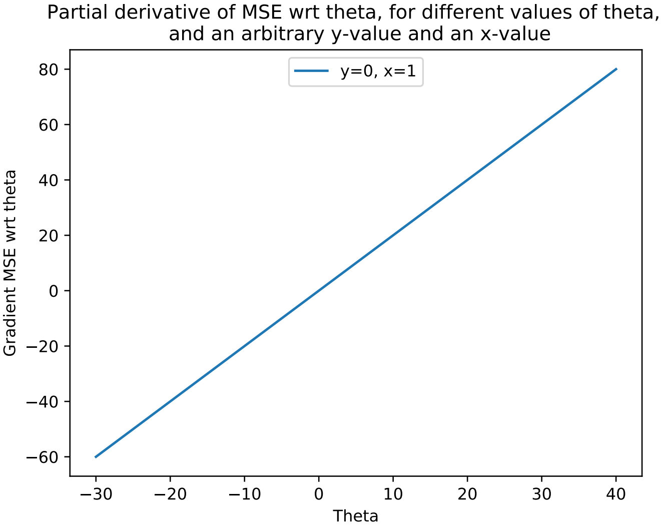
Mean Squared Error



Gradient of the Mean Squared Error wrt the weights

```
#note: assume n = 1
import matplotlib.pyplot as plt
import numpy as np
theta = np.arange(-30,41)
x = 1
yhat = theta * x
y = 0 #true value is arbitrarily chosen to be 0
mse_grad = 2*(yhat - y)*x
plt.plot(theta, mse_grad, label = 'y=0, x=1')
plt.legend(loc="upper center")
plt.title('Partial derivative of MSE wrt theta, for different
\n \
values of theta, and an arbitrary y-value and an x-value')
plt.xlabel('Theta')
plt.ylabel('Gradient MSE wrt theta')
plt.savefig('msegrad.pdf')
```

Gradient of the Mean Squared Error wrt the weights



Mean Absolute Error

Another popular loss function is the MAE.

$$\begin{aligned} MAE(y, h_\theta(X)) &= \frac{1}{n}\sum_{i=1}^{n} |h_\theta(X_i) - y_i| \\ \frac{\partial MAE}{\partial \theta_j} &= \frac{1}{n}\sum_{i=1}^{n} \frac{h_\theta(X_i) - y_i}{|h_\theta(X_i) - y_i|}\frac{\partial h_\theta(X_i)}{\partial \theta_j} \\ &= \frac{1}{n}\sum_{i=1}^{n} \frac{\hat{y}_i - y_i}{|\hat{y}_i - y_i|}\frac{\partial \hat{y}_i}{\partial \theta_j} \\ &= \frac{1}{n}\sum_{i=1}^{n} sign(\hat{y}_i - y_i)\frac{\partial \hat{y}_i}{\partial \theta_j} \end{aligned} \tag{14}$$

Specifically, for the weight vector θ connecting the last hidden layer and the output layer:

$$\begin{aligned} MAE(y, h_\theta(X)) &= \frac{1}{n}\sum_{i=1}^{n} |X_i\theta - y_i| \\ \frac{\partial MAE}{\partial \theta_j} &= \frac{1}{n}\sum_{i=1}^{n} sign(X_i\theta - y_i)X_{i,j} \end{aligned} \tag{15}$$

Mean Absolute Error

Advantages:

- Robust against outliers

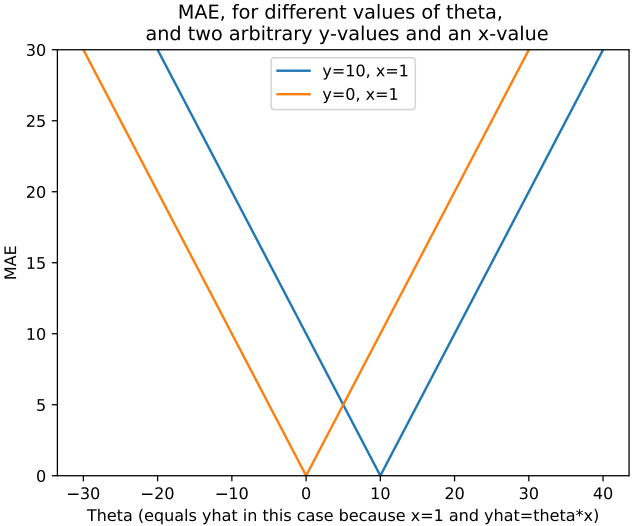
Disadvantages:

- Slower learning because predictions farther away from the true point result in a proportionate loss. For example, being 5 points off from the true point results in a loss of 5, whereas being 10 points off (i.e., 2 times 5) results in a loss of 10 (2 times 5).
- Harder to converge to minimum because the gradient is constant (either -1 or 1 when x = 1) and thus as we get closer to the minimum we will be bouncing around the minimum.
- The gradient is also undefined when the $\hat{y} - y = 0$ because $\frac{\hat{y}_i - y_i}{|\hat{y}_i - y_i|} \frac{\partial \hat{y}_i}{\partial \theta_j} = \frac{0}{|0|} \frac{\partial \hat{y}_i}{\partial \theta_j}$. When this happens we use 0 for the gradient. It does not happen often that the loss is 0.

Mean Absolute Error

```
#note: assume n = 1
import matplotlib.pyplot as plt
import numpy as np
#loss plot for two artibrarily chosen y-values and an x-value
#and different values of theta
theta = np.arange(-30,41)
x = 1
yhat = theta * x

y = 10 #true value is arbitrarily chosen to be 10
mae = np.absolute(y - yhat)
plt.plot(yhat,mae, label = 'y=10, x=1')

y = 0 #true value is arbitrarily chosen to be 0
mae = np.absolute(y - yhat)
plt.plot(yhat,mae,label = 'y=0, x=1')

plt.legend(loc="upper center")
plt.ylim(0, 30)
plt.title('MAE, for different values of theta, \n \
and two arbitrary y-values and an x-value')
plt.xlabel('Theta (equals yhat in this case because x=1 and \n \
yhat=theta*x)')
plt.ylabel('MAE')
plt.savefig('mae.pdf')
```

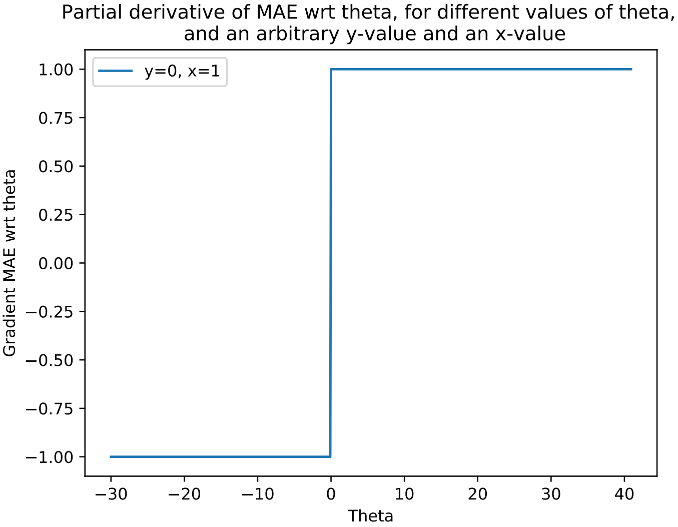
Mean Absolute Error



Gradient of the Mean Absolute Error wrt the weights

```
#note: assume n = 1, and a linear model
import matplotlib.pyplot as plt
import numpy as np
theta = np.arange(-30,41,0.1)
x = 1
yhat = theta * x
y = 0 #true value is arbitrarily chosen to be 0
mae_grad = np.sign(yhat - y)*x
plt.plot(theta, mae_grad, label = 'y=0, x=1')
plt.legend(loc="upper left")
plt.title('Partial derivative of MAE wrt theta, for different
\n \
values of theta, and an arbitrary y-value and an x-value')
plt.xlabel('Theta')
plt.ylabel('Gradient MAE wrt theta')
plt.savefig('maegrad.pdf')
```

Gradient of the Mean Squared Error wrt the weights



Huber Loss

The Huber Loss is defined as

$$HuberLoss(y, h_\theta(X)) = \frac{1}{n}\sum_{i=1}^{n}\begin{cases}\frac{1}{2}(h_\theta(X_i) - y_i)^2 & |h_\theta(X_i) - y_i| \leq \delta \\ \delta(|h_\theta(X_i) - y_i| - \frac{1}{2}\delta) & otherwise\end{cases},$$

where $\delta \in \mathbb{R}$. The partial derivative

$$\frac{\partial HuberLoss}{\partial \theta_j} = \frac{1}{n}\sum_{i=1}^{n}\begin{cases}(h_\theta(X_i) - y_i)\frac{\partial h_\theta}{\partial \theta_j} & |h_\theta(X_i) - y_i| \leq \delta \\ \delta\frac{h_\theta(X_i)-y_i}{|h_\theta(X_i)-y_i|}\frac{\partial h_\theta}{\partial \theta_j} & otherwise\end{cases},$$

where $h_\theta(X_i)$ is $\hat{y}_i$ and we use $sign(h_\theta(X_i) - y_i)$ for $\frac{h_\theta(X_i)-y_i}{|h_\theta(X_i)-y_i|}$.

Huber Loss

Making the equations in the previous slide more specific, assume we are talking about the weight vector θ connecting the last hidden layer and the output layer, we get

$$HuberLoss(y, h_\theta(X)) = \frac{1}{n}\sum_{i=1}^{n}\begin{cases}\frac{1}{2}(X_i\theta - y_i)^2 & |X_i\theta - y_i| \leq \delta \\ \delta(|X_i\theta - y_i| - \frac{1}{2}\delta) & otherwise\end{cases},$$

where $\delta \in \mathbb{R}$. The partial derivative

$$\frac{\partial HuberLoss}{\partial \theta_j} = \frac{1}{n}\sum_{i=1}^{n}\begin{cases}(X_i\theta - y_i)X_{i,j} & |X_i\theta - y_i| \leq \delta \\ \delta\frac{X_i\theta - y_i}{|X_i\theta - y_i|}X_{i,j} & otherwise\end{cases},$$

where $X_i\theta$ is $\hat{y}_i$ and we use $sign(X_i\theta - y_i)$ for $\frac{X_i\theta - y_i}{|X_i\theta - y_i|}$.

Huber Loss

Advantages:

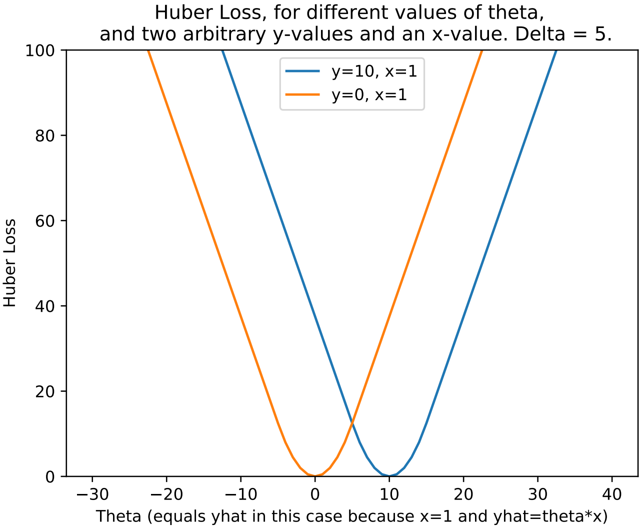
- Robust against outliers
- No undefined gradients
- Easy to converge to minimum because as we get closer to the minimum (and when absolute difference is smaller than δ) the gradient becomes increasingly small (as opposed to constant); no bouncing around the minimum.

Disadvantages:

- For absolute differences greater than δ we have slower learning because predictions farther away from the true point result in a proportionate loss. For example, being 5 points off from the true point results in a loss of 5, whereas being 10 points off (i.e., 2 times 5) results in a loss of 10 (2 times 5).

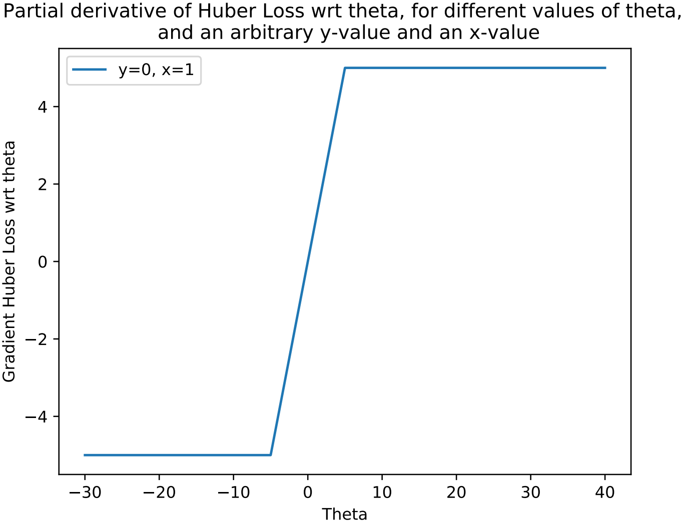
Huber Loss

for a linear model

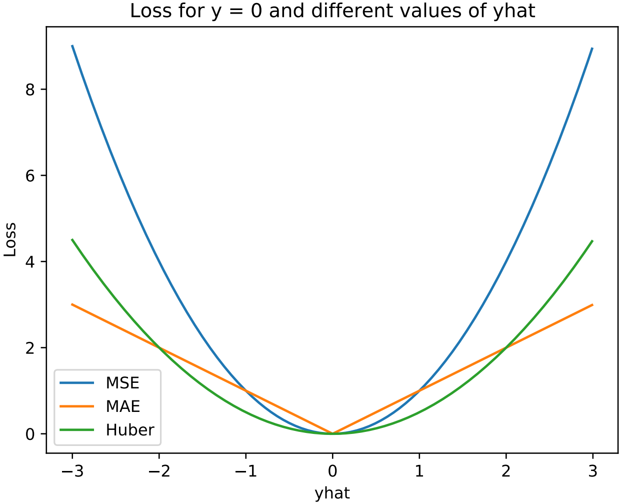


Gradient of Huber Loss wrt weights

for a linear model



MSE, MAE and Huber Loss compared



Other ways to be robust against outliers

- Instead of using the Huber Loss, we could also just use the MSE (and benefit from all its advantages) and clip gradients that are too large.
- This strategy offers more control, because it allows us to set a maximum gradient. The gradient of the MAE is smaller than the gradient of the MSE but can still be very large, so using the MAE instead of the MSE helps but is not guaranteed to keep the gradient in check.
- I have personally used the MSE with gradient clipping with great success in many projects.
- We will see gradient clipping later.

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables

- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables

- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions

- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Outline I

- A note on the technology used in this class: TensorFlow (TF)
- Start of field of Neural Networks: the Perceptron
- Continuous response variables
  - Architecture for a single continuous response
  - Architecture for multiple continuous responses
  - Loss functions for (multiple) continuous response
- Categorical response variables
  - Architecture for binary response
  - Output activation function for binary response
  - Loss function for binary response
  - Architecture for multi-label response
  - Output activation function for multi-label response
  - Loss function for multi-label response
  - Architecture for multi-class response
  - Output activation function for multi-class response
  - Loss function for multi-class response
  - Review Multi-label vs Multi-class
- Activation functions
  - What is an activation function, and do we need it?
  - What does a good activation function look like?
  - Sigmoid
  - Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Architecture for binary response

- Single output neuron
- Hidden nodes/layers: use sigmoid activation for now, we will see a large variety of activation functions in a subsequent session. Any number of hidden layers and neurons.
- Output activation is sigmoid. This will squash the value between $[0, 1]$ and it can be interpreted as a probability of the positive class. The estimated probability of the negative class is $1 - \mathit{probability\ of\ positive\ class}$.
- Loss: cross-entropy (also called log loss)
- See **BinaryClassificationTF.py**

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables
- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables
- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions
- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

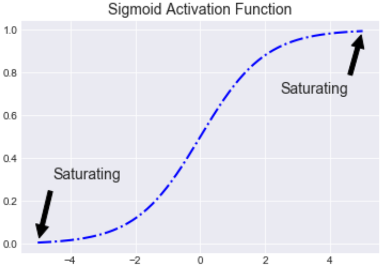
- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

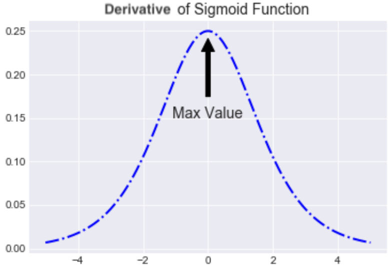
Output activation function for binary response

Sigmoid Function

The sigmoid function is one of the mostly-used and well-known activation functions.

$$sigmoid(x) = \frac{e^x}{1+e^x} = \frac{\frac{e^x}{e^x}}{\frac{1}{e^x}+\frac{e^x}{e^x}} = \frac{1}{1+e^{-x}} \tag{16}$$





Output activation function for binary response

Sigmoid Function

$$
\begin{aligned}
\frac{\partial}{\partial x} sigmoid(x) &= \frac{\partial}{\partial x}\left(\frac{1}{1+e^{-x}}\right) \\
&= \frac{-\frac{\partial}{\partial x}(1+e^{-x})}{(1+e^{-x})^2} && \#\text{reciprocal rule} \\
&= \frac{e^{-x}}{(1+e^{-x})^2} && \#\text{chain rule} \\
&= \frac{e^{-x}+1-1}{(1+e^{-x})^2} && \#\text{add and subtract 1} \\
&= \frac{e^{-x}+1}{(1+e^{-x})^2} - \left(\frac{1}{1+e^{-x}}\right)^2 \\
&= \frac{1}{1+e^{-x}} - \left(\frac{1}{1+e^{-x}}\right)^2 && \#\text{divide } 1^{st} \text{ term by } 1+e^{-x} \\
&= sigmoid(x) - sigmoid^2(x) \\
&= sigmoid(x)\,(1 - sigmoid(x))
\end{aligned}
\tag{17}
$$

Outline I

- A note on the technology used in this class: TensorFlow (TF)
- Start of field of Neural Networks: the Perceptron
- Continuous response variables
    - Architecture for a single continuous response
    - Architecture for multiple continuous responses
    - Loss functions for (multiple) continuous response
- Categorical response variables
    - Architecture for binary response
    - Output activation function for binary response
    - Loss function for binary response
    - Architecture for multi-label response
    - Output activation function for multi-label response
    - Loss function for multi-label response
    - Architecture for multi-class response
    - Output activation function for multi-class response
    - Loss function for multi-class response
    - Review Multi-label vs Multi-class
- Activation functions
    - What is an activation function, and do we need it?
    - What does a good activation function look like?
    - Sigmoid
    - Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Loss function for binary response

Cross Entropy

In the previous slides we have seen how the neural network makes predictions that fall into $[0, 1]$. The question now becomes what the objective function is and how the network is trained. The neural network will adjust the parameters θ so that the model predicts high probabilities for positive instances (y = 1) and low probabilities for negative instances (y=0). This is captured by the loss function (for a single instance):

$$c(\theta) = \begin{cases} -log(\hat{y}) & \text{if } y = 1 \\ -log(1 - \hat{y}) & \text{if } y = 0. \end{cases} \tag{18}$$

We can rewrite this and also average across all instances:

$$c(\theta) = -\frac{1}{n} \sum_{i=1}^{n} \left[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i) \right] \tag{19}$$

This loss makes sense when we look at a plot of the log function:

- $-log(\hat{y})$ is very large (small) when $\hat{y}$ approaches 0 (1)
- $-log(1 - \hat{y})$ is very small (large) when $\hat{y}$ approaches 0 (1)

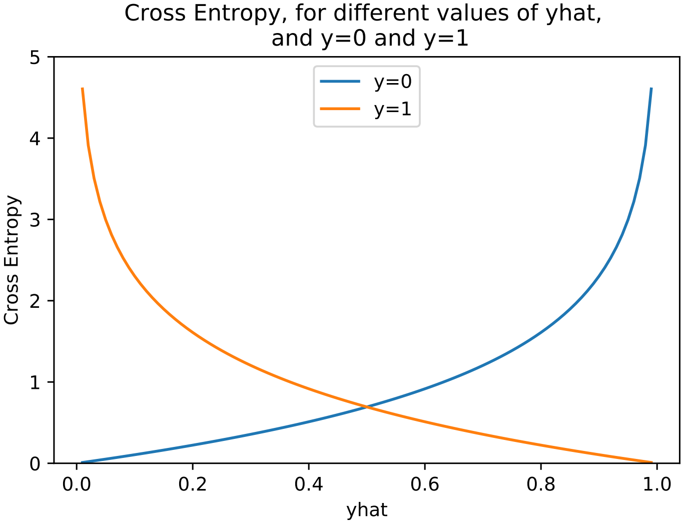
Loss function for binary response

Cross Entropy

```
#note: n = 1
import matplotlib.pyplot as plt
import numpy as np
#loss plot for y=0 and y=1
#and different values of yhat
#(or different values of theta and value for x, e.g., x = 1)
yhat = np.arange(0.01,1.0,0.01)

y = 0 #true value is 0
ce = - (y*np.log(yhat) + (1-y)*np.log(1-yhat)) #for n=1
plt.plot(yhat,ce, label = 'y=0')

y = 1 #true value is 1
ce = - (y*np.log(yhat) + (1-y)*np.log(1-yhat)) #for n=1
plt.plot(yhat,ce,label = 'y=1')

plt.legend(loc="upper center")
plt.ylim(0, 5)
plt.title('Cross Entropy, for different values of yhat, \n \
and y=0 and y=1')
plt.xlabel('yhat')
plt.ylabel('Cross Entropy')
plt.savefig('ce.pdf')
```

Loss function for binary response

Cross Entropy



Cross-Entropy

We can use (stochastic) gradient descent to find θ. The partial derivatives of the cross entropy function preceded by the sigmoid with regard to the j^{th} parameter θ_j is obtained as follows. Assume we are talking about the weight vector θ connecting the last hidden layer and the output layer.

Remember:

$$\begin{aligned} z_i &= X_i\theta \\ \hat{y}_i &= sigmoid(z_i) \\ c_i(\theta) &= -\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right] \end{aligned} \tag{20}$$

Cross-Entropy

Derivative of cross-entropy ($\hat{y}$ is activated weighted sum, z is non-activated weighted sum, i indicates an instance, θ_j is a weight):

$$\frac{\partial c_i}{\partial \theta_j} = \frac{\partial c_i}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial z_i}\frac{\partial z_i}{\partial \theta_j} \tag{21}$$

Examining each factor in turn:

$$\begin{aligned}
\frac{\partial c_i}{\partial \hat{y}_i} &= -(\frac{y_i}{\hat{y}_i} + \frac{1-y_i}{1-\hat{y}_i}(0-1)) && \#\text{chain rule} \\
&= -(\frac{y_i}{\hat{y}_i} - \frac{1-y_i}{1-\hat{y}_i}) && \#\text{rewrite} \\
&= -\frac{y_i(1-\hat{y}_i)}{\hat{y}_i(1-\hat{y}_i)} + \frac{\hat{y}_i(1-y_i)}{\hat{y}_i(1-\hat{y}_i)} && \#\text{set same denom.} \\
&= \frac{\hat{y}_i - y_i}{\hat{y}_i(1-\hat{y}_i)} && \#\text{simplify} \\
\frac{\partial \hat{y}_i}{\partial z_i} &= \hat{y}_i(1-\hat{y}_i) && \#\text{deriv. of sigmoid} \\
\frac{\partial z_i}{\partial \theta_j} &= X_{i,j}
\end{aligned} \tag{22}$$

Cross-Entropy

Combining things:

$$\begin{aligned} \frac{\partial c_i}{\partial \theta_j} &= (\hat{y}_i - y_i)X_{i,j} \\ \frac{\partial c}{\partial \theta_j} &= \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)X_{i,j} \end{aligned} \tag{23}$$

Cross-Entropy

This result means that

- **(case 1)** the derivative of the combination of the linear activation function and the $\frac{1}{2}MSE$ loss (see Equation 13 for the derivative of the MSE combined with the linear activation function) wrt the weights, and
- **(case 2)** the derivative of the combination of the sigmoid activation function and the cross-entropy loss (Equation 23) wrt the weights

look identical; the partial derivative wrt θ_j is $\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)X_{i,j}$ in both cases.

However, we shouldn't be fooled by this statement. While at the surface this looks identical, the $\hat{y}_i$ values are different depending on the case. In case 1 above, $\hat{y}_i = X_i\theta$. In case 2 above, $\hat{y}_i = \frac{1}{1+e^{-X_i\theta}}$. In sum:

$$\frac{1}{n}\sum_{i=1}^{n}(X_i\theta - y_i)X_{i,j} \neq \frac{1}{n}\sum_{i=1}^{n}(\frac{1}{1+e^{-X_i\theta}} - y_i)X_{i,j} \tag{24}$$

Why not sigmoid and MSE?

Comparison

Why can we not use MSE in the classification setting?

- First, note that we need to guarantee that the values are in $[0, 1]$, and thus we need the sigmoid.
- It turns out that the combination of the sigmoid and MSE results in a non-convex problem (see next slides).
- Linking to the discussion on the previous slide, we are not comparing 'sigmoid & cross entropy' and 'linear & 1/2MSE', but 'sigmoid & cross entropy' and 'sigmoid & 1/2MSE'.
- Let's make that comparison on the next slide.

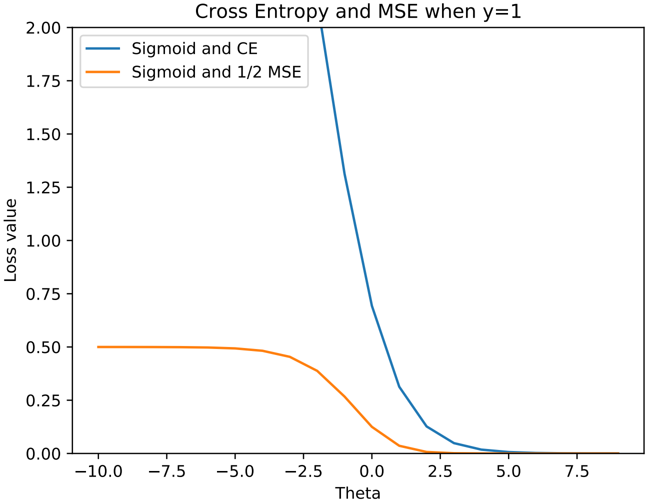
Why not sigmoid and MSE?

Comparison

```python
import numpy as np
import matplotlib.pyplot as plt
y = 1 #true value is 0
x = 1
theta = np.arange(-10,10,1)
z = theta*x
yhat = 1 / (1 + np.exp(-z))
#CE
ce = - ( y*np.log(yhat) + (1-y)*np.log(1-yhat) )
#MSE
halfmse = ((yhat - y)**2)/2
#plot
plt.plot(theta,ce,label = 'Sigmoid and CE')
plt.plot(theta,halfmse,label = 'Sigmoid and 1/2 MSE')
plt.legend(loc="upper left")
plt.ylim(0,2)
plt.title('Cross Entropy and MSE when y=1')
plt.xlabel('Theta')
plt.ylabel('Loss value')
plt.savefig('comparison.pdf')
```

Why not sigmoid and MSE?

Comparison



Why not sigmoid and MSE?

Comparison

- We cannot use the sigmoid and the MSE together because the MSE is non-convex in this case.
- This means it is not guaranteed that we will minimize the loss.
- To see this, consider what happens when the network is initialized in such a way that $\theta < -5$. The curve is flat there, and so the derivative will be zero, or close to it. This means that there will be no updates to θ, or the updates will be very slow.

In sum, we should not use the sigmoid activation function together with the MSE. We need the sigmoid in classification to guarantee that the outputs will fall in $[0, 1]$.

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables

- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables

- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions

- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Architecture for multi-label response

- Multiple independent response variables
- One output neuron per label
- Hidden nodes/layers: same as for binary response (use sigmoid for now)
- Output activation is sigmoid for each output separately
- Loss: cross-entropy
- See **MultiLabelClassificationTF.py**

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables

- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables

- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions

- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

Outline II

ReLU
Leaky ReLU
PReLU
ELU
Softplus
Summary

Gradient Descent
Notation
Manual implementation of Gradient Descent
Sensitivity to learning rate
Learning Rate
Gradient Descent vs Other optimizers
Gradient Descent Variants
Batch Gradient Descent
Stochastic Gradient Descent Algorithm
Mini-batch Gradient Descent Algorithm

Output activation function for multi-label response

- Exactly the same as for the binary case, as multi-label is equivalent to learning from multiple independent binary responses

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables

- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables

- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions

- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Loss function for multi-label response

- Exactly the same as for the binary case, as multi-label is equivalent to learning from multiple independent binary responses, with the addition that
- the cross-entropy is averaged across all labels.

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables
- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables
- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions
- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Architecture for multi-class response

- Each instance can belong to only one class (i.e., classes are mutually exclusive) out of three or more classes
- One output neuron per class
- Hidden nodes/layers: same as other architectures (use sigmoid for now)
- Output activation is softmax (combines all outputs)
- Loss: cross entropy
- See **PreprocessYforMultiClass.py**
- See **MultiClassClassificationTF.py**

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables

- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables

- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions

- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

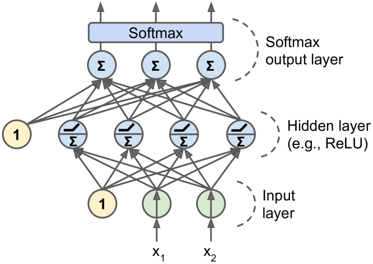
Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Output activation function for multi-class response

Softmax Function

The softmax activation function (also called normalized exponential) generalizes the sigmoid function to multiple classes. It takes the non-activated weighted sum of each of the output variables (each representing a class), which may be negative or greater than one, and may not add up to one, and ensures each of the values are then a value in [0,1] and add up to 1. This allows us to interpret them as probabilities.



https://www.oreilly.com/library/view/neural-networks-and/9781492037354/ch01.html

Output activation function for multi-class response

Softmax Function

Remember the sigmoid function:

$$P(Y=1|W,b;X)=\frac{e^{(XW+b)}}{1+e^{(XW+b)}}=\frac{1}{1+e^{-(XW+b)}}. \tag{25}$$

This is the softmax function, for K classes:

$$\begin{bmatrix} P(Y=1|W_1,b_1;X) \\ P(Y=2|W_2,b_2;X) \\ \vdots \\ P(Y=K|W_k,b_k;X) \end{bmatrix} = \frac{1}{\sum_{j=1}^{K} e^{(XW_j+b_j)}} \begin{bmatrix} e^{(XW_1+b_1)} \\ e^{(XW_2+b_2)} \\ \vdots \\ e^{(XW_K+b_K)} \end{bmatrix}, \tag{26}$$

where $W_1, W_2, ..., W_K$ represent dedicated parameter vectors per output class, and $b_1, b_2, ..., b_K$ represent a dedicated bias per output class. **Softmax computes the exponential of every un-activated output (what you see in the square brackets on the right hand side of the equation), and then divides by the sum of all the exponentials.**

Output activation function for multi-class response

Softmax Function

Can you show how the softmax is a general version of the sigmoid? Consider K = 2. The softmax is then written as follows:

$$\begin{aligned}\begin{bmatrix} P(Y=1|W_1,b_1;X) \\ P(Y=2|W_2,b_2;X) \end{bmatrix} &= \frac{1}{\sum_{j=1}^{2} e^{(XW_j+b_j)}} \begin{bmatrix} e^{(XW_1+b_1)} \\ e^{(XW_2+b_2)} \end{bmatrix} \\ &= \frac{1}{e^{(XW_1+b_1)}+e^{(XW_2+b_2)}} \begin{bmatrix} e^{(XW_1+b_1)} \\ e^{(XW_2+b_2)} \end{bmatrix} \end{aligned} \tag{27}$$

Note that this equation is overparameterized in that we do not need to estimate W2 and b2 (or W1 and b1), because P(Y = 2) = 1-P(Y=1). They are redundant. So, let's take out W2 and b2 in the following slide.

Output activation function for multi-class response

Softmax Function

Taking out W2 and b2 by subtraction:

$$
\begin{aligned}
\begin{bmatrix} P(Y=1) \\ P(Y=2) \end{bmatrix} &= \frac{1}{e^{(X(W_1-W_2)+b_1-b_2)}+e^{(0)}} \begin{bmatrix} e^{(X(W_1-W_2)+b_1-b_2)} \\ e^{(0)} \end{bmatrix} \\
&= \begin{bmatrix} \frac{e^{(X(W_1-W_2)+b_1-b_2)}}{e^{(X(W_1-W_2)+b_1-b_2)}+1} \\ \frac{1}{e^{(X(W_1-W_2)+b_1-b_2)}+1} \end{bmatrix} \\
&= \begin{bmatrix} \frac{e^{(X(W_1-W_2)+b_1-b_2)}+1}{e^{(X(W_1-W_2)+b_1-b_2)}+1} - \frac{1}{e^{(X(W_1-W_2)+b_1-b_2)}+1} \\ \frac{1}{e^{(X(W_1-W_2)+b_1-b_2)}+1} \end{bmatrix} \\
&= \begin{bmatrix} 1-\frac{1}{1+e^{(X(W_1-W_2)+b_1-b_2)}} \\ \frac{1}{1+e^{(X(W_1-W_2)+b_1-b_2)}} \end{bmatrix}
\end{aligned} \qquad (28)
$$

Output activation function for multi-class response

Softmax Function

Clearly there is no need to be estimating both W_1, b_1 and W_2, b_2. Next, to see the equivalence with the sigmoid, set W_2-W_1 to W, and note that (W_1 - W_2) = - (W_2-W_1) = - W. Also set b_2-b_1 to b, and note that b_1 - b_2 = - (b_2-b_1) = - b.

$$\begin{bmatrix} 1 - \dfrac{1}{1 + e^{(X(W_1 - W_2) + b_1 - b_2)}} \\ \dfrac{1}{1 + e^{(X(W_1 - W_2) + b_1 - b_2)}} \end{bmatrix} = \begin{bmatrix} 1 - \dfrac{1}{1 + e^{(X(-W) - b)}} \\ \dfrac{1}{1 + e^{(X(-W) - b)}} \end{bmatrix} \tag{29}$$

Note: the word softmax comes from the word argmax. Example: argmax([0,2,1]) = [0,1,0] (i.e., the second position contains the maximum) and softmax([0,2,1]) = [0.09003057, 0.66524096, 0.24472847]. It is soft because it is continuous and differentiable.

Outline I

- A note on the technology used in this class: TensorFlow (TF)
- Start of field of Neural Networks: the Perceptron
- Continuous response variables
    - Architecture for a single continuous response
    - Architecture for multiple continuous responses
    - Loss functions for (multiple) continuous response
- Categorical response variables
    - Architecture for binary response
    - Output activation function for binary response
    - Loss function for binary response
    - Architecture for multi-label response
    - Output activation function for multi-label response
    - Loss function for multi-label response
    - Architecture for multi-class response
    - Output activation function for multi-class response
    - Loss function for multi-class response
    - Review Multi-label vs Multi-class
- Activation functions
    - What is an activation function, and do we need it?
    - What does a good activation function look like?
    - Sigmoid
    - Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Loss function for multi-class response

Cross-Entropy

This is a generalization to multiple classes from the binary case we saw earlier. Cross entropy for a single instance:

$$c(\theta) = -\sum_{k=1}^{K} y_{i,k} \log \hat{y}_{i,k} \tag{30}$$

For all instances:

$$c(\theta) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{k=1}^{K} y_{i,k} \log \hat{y}_{i,k} \tag{31}$$

where $y_{i,k}$ is equal to 1 if the target class for instance i is class k; otherwise it is equal to 0. In other words, we are multiplying 1 by the log of the probability of the target class and discarding everything else. This means that only the probability on the target class matters.

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables

- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables

- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions

- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

Outline II

ReLU
Leaky ReLU
PReLU
ELU
Softplus
Summary

Gradient Descent

Notation
Manual implementation of Gradient Descent
Sensitivity to learning rate
Learning Rate
Gradient Descent vs Other optimizers
Gradient Descent Variants
Batch Gradient Descent
Stochastic Gradient Descent Algorithm
Mini-batch Gradient Descent Algorithm

Review Multi-label vs Multi-class

	Multi-label	Multi-class
instance can belong to	more than one class (label) out of two or more classes	a single class out of three or more classes
output neurons	one per label	one per class
output activation	sigmoid (separately to each output)	softmax (jointly to all outputs)
loss function	cross entropy	cross entropy
loss function name in TF	binary_crossentropy	categorical_crossentropy
sparse loss function name in TF	None	sparse_categorical_crossentropy

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables

- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables

- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions

- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

Outline II

ReLU
Leaky ReLU
PReLU
ELU
Softplus
Summary

Gradient Descent

Notation
Manual implementation of Gradient Descent
Sensitivity to learning rate
Learning Rate
Gradient Descent vs Other optimizers
Gradient Descent Variants
Batch Gradient Descent
Stochastic Gradient Descent Algorithm
Mini-batch Gradient Descent Algorithm

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables

- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables

- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions

- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

What is an activation function?

- A neuron calculates a ‘weighted sum’ of its inputs, and adds a bias.
- This weighted sum can be anything ranging from negative infinity to positive infinity. The number may be too precise or large to be represented, which results in either numerical underflow or numerical overflow. The neuron really does not know the bounds of the value.
- The neuron then needs to decide whether it should be ‘fired’ or not (analogy comes from our brains’ neurons).
- This decision is made by a function. This function thus decides whether a neuron should be activated or not (hence, the name ‘activation’ function). In other words it decides whether the information that the neuron is receiving is relevant or whether it should be ignored.

Can we do without an activation function?

- Not having an activation function is equivalent to having a linear transformation.
- A linear equation is simple to solve but is limited in its capacity to solve complex non-linear problems.
- A neural network without an activation function is essentially just a linear model. Adding additional layers will not improve the capacity of the model.
- The purpose of the activation function is to introduce non-linearities into the output of a neuron, thereby making it capable to learn and perform more complex tasks.
- We want our neural networks to work on complex tasks such as language translation and image classification. Linear transformations would never be able to perform such tasks.

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables
- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables
- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions
- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

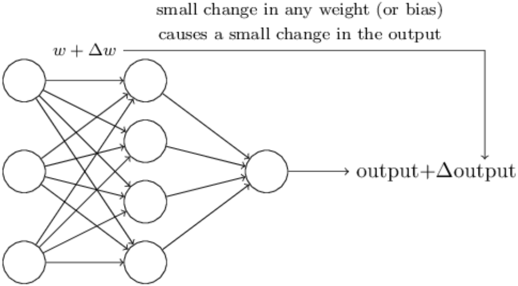
Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

What does a good activation function look like?

Smooth.

This is what we want (to avoid weight updates that are too large, and potentially result in divergent learning):



small change in any weight (or bias)
causes a small change in the output
$w + \Delta w$
output+Δoutput

http://neuralnetworksanddeeplearning.com/chap1.html

What does a good activation function look like?

- The reason why we want a small change in any parameter (w or b) to cause a small change in the output is because we can then use that fact to change the parameters in the direction so that the output gets closer to the target.
- If a small change in a parameter would result in a dramatic change in the output, it becomes really hard to gradually modify the network so that it gets closer to the desired behavior.
- It turns out that the step (heaviside) activation function creates these large changes when a small change in the weights or biases happens. Therefore we need to modify the step function to be smoother (differentiable), so that a small change in the weights or biases cannot result in a drastic change in the output. One function that does that is the sigmoid function. We'll look at alternatives in the next batch of slides.

What does a good activation function look like?

- This realization was a major breakthrough in neural network research and it cleared the path for us to measure the small changes using partial derivatives (the step function has only flat segments: the derivative is either 0 or non-defined (infinite slope)) and so the backpropagation algorithm was born (1986, Rumelhart et al.).
- Backpropagation uses derivatives to measure a change in the loss function with respect to a change in the weight, and updates the weights. The latter is called the Gradient Descent step.
- More specifically, backpropagation works as follows. For each training instance the neural net makes a prediction (forward pass), computes the error, and then goes in reverse to measure how much each weight and bias contributed to the error (using partial derivatives). Finally, it updates the weights and biases to reduce the error using an update rule very similar to the perceptron update rule. More about backpropagation and Gradient Descent soon.

Outline I

- A note on the technology used in this class: TensorFlow (TF)
- Start of field of Neural Networks: the Perceptron
- Continuous response variables
  - Architecture for a single continuous response
  - Architecture for multiple continuous responses
  - Loss functions for (multiple) continuous response
- Categorical response variables
  - Architecture for binary response
  - Output activation function for binary response
  - Loss function for binary response
  - Architecture for multi-label response
  - Output activation function for multi-label response
  - Loss function for multi-label response
  - Architecture for multi-class response
  - Output activation function for multi-class response
  - Loss function for multi-class response
  - Review Multi-label vs Multi-class
- Activation functions
  - What is an activation function, and do we need it?
  - What does a good activation function look like?
  - Sigmoid
  - Tanh

Outline II

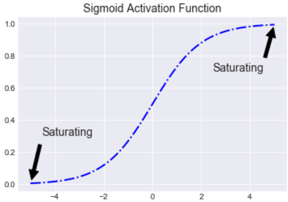
ReLU
Leaky ReLU
PReLU
ELU
Softplus
Summary

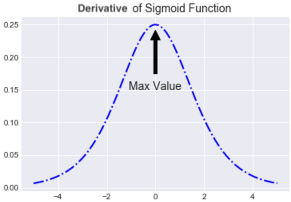
Gradient Descent

Notation
Manual implementation of Gradient Descent
Sensitivity to learning rate
Learning Rate
Gradient Descent vs Other optimizers
Gradient Descent Variants
Batch Gradient Descent
Stochastic Gradient Descent Algorithm
Mini-batch Gradient Descent Algorithm

Sigmoid

- See the subsection ‘Output activation function for binary response’: Equation 16 and subsequent derivations.
- Advantages:
  - Smooth function and gradient.
  - Activations are bounded.
  - Squashes output to $[0, 1]$: great choice for output layer in classification problems.
- Disadvantages:
  - Extreme x values will result in near-zero gradients; the gradient is said to vanish and this slows down training.





```
tf.keras.layers.Dense(n_units, activation = 'sigmoid', ...)
```

Outline I

- A note on the technology used in this class: TensorFlow (TF)
- Start of field of Neural Networks: the Perceptron
- Continuous response variables
  - Architecture for a single continuous response
  - Architecture for multiple continuous responses
  - Loss functions for (multiple) continuous response
- Categorical response variables
  - Architecture for binary response
  - Output activation function for binary response
  - Loss function for binary response
  - Architecture for multi-label response
  - Output activation function for multi-label response
  - Loss function for multi-label response
  - Architecture for multi-class response
  - Output activation function for multi-class response
  - Loss function for multi-class response
  - Review Multi-label vs Multi-class
- Activation functions
  - What is an activation function, and do we need it?
  - What does a good activation function look like?
  - Sigmoid
  - Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

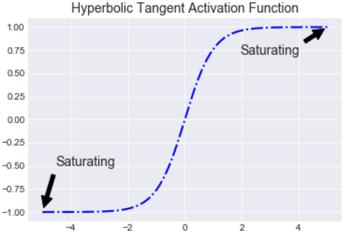
Gradient Descent

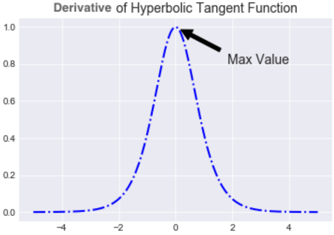
- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Hyperbolic Tangent Function

Another popular activation function is the Hyperbolic Tangent (tanh) function and is the ratio of the hyperbolic sine and hyperbolic cosine. It is defined as follows:

$$
\begin{aligned}
tanh(x) &= \frac{sinh(x)}{cosh(x)} = \frac{\frac{e^x - e^{-x}}{2}}{\frac{e^x + e^{-x}}{2}} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{\frac{e^x(e^x - e^{-x})}{e^x}}{\frac{e^x(e^x + e^{-x})}{e^x}} \\
&= \frac{e^x(e^x - e^{-x})}{e^x(e^x + e^{-x})} = \frac{e^{2x} - \frac{e^x}{e^x}}{e^{2x} + \frac{e^x}{e^x}} = \frac{e^{2x} - 1}{e^{2x} + 1}
\end{aligned}
\tag{32}
$$





Hyperbolic Tangent Function

How is the tanh a rescaled version of the sigmoid function?
Remember:

$$\begin{aligned} tanh(x) &= \frac{e^{2x}-1}{e^{2x}+1} \\ sigmoid(x) &= \frac{e^{x}}{1+e^{x}} \end{aligned} \tag{33}$$

Tanh is rescaled version of sigmoid:

$$\begin{aligned} 2sigmoid(2x)-1 &= 2\frac{e^{2x}}{1+e^{2x}} - 1 \\ &= \frac{e^{2x}}{1+e^{2x}} + \frac{e^{2x}}{1+e^{2x}} - \frac{1+e^{2x}}{1+e^{2x}} \\ &= \frac{e^{2x}+e^{2x}-1-e^{2x}}{1+e^{2x}} \\ &= \frac{e^{2x}-1}{1+e^{2x}} = \frac{e^{2x}-1}{e^{2x}+1} = tanh(x) \end{aligned} \tag{34}$$

Hyperbolic Tangent Function

$$\begin{aligned}
\frac{\partial tanh(x)}{\partial x} &= \frac{\partial \frac{sinh(x)}{cosh(x)}}{\partial x} \\
&= \frac{\frac{\partial sinh(x)}{\partial x} cosh(x) - \frac{\partial cosh(x)}{\partial x} sinh(x)}{cosh^2(x)} && \#\text{quotient rule} \\
& && \#(f/g)' = (f'g - g'f)/g^2 \\
&= \frac{cosh^2(x) - sinh^2(x)}{cosh^2(x)} && \#\text{sinh(x)' = cosh(x)} \\
& && \#\text{cosh(x)' = sinh(x)} \\
&= 1 - \frac{sinh^2(x)}{cosh^2(x)} \\
&= 1 - tanh^2(x) \\
&= 1 - (tanh(x))^2
\end{aligned} \tag{35}$$

Hyperbolic Tangent Function

```
tf.keras.layers.Dense(n_units, activation = 'tanh', ...)
```

- Advantages:
  - Same as sigmoid
  - Gradient is stronger than sigmoid (derivatives are steeper)
  - Center of activation is 0, making it easier for subsequent layers to stay away from extremities of the function where saturation happens.
- Disadvantages:
  - Also has the vanishing gradient problem

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables

- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables

- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions

- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

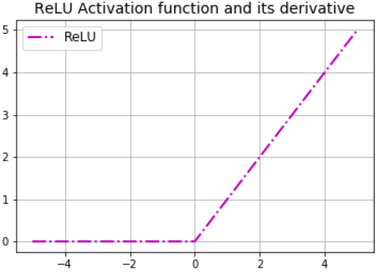
Gradient Descent

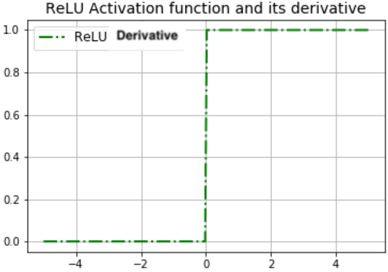
- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Rectified Linear Unit (ReLU)

ReLU is a piecewise linear function that corresponds to:

$$ReLU(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \tag{36}$$





```
tf.keras.layers.Dense(n_units, activation = 'relu', ...)
```

Rectified Linear Unit (ReLU)

The term 'Rectified' comes from an electrical device, a rectifier that converts alternating current (AC), which periodically reverses direction, to direct current (DC), which flows in only one direction. The process is known as rectification, since it straightens the direction of current. Another way of writing the ReLU function is:

$$ReLU(x) = max(0, x) \tag{37}$$

Example for x=-2, ReLu = max(0,-2) = 0
Example for x=2, ReLu = max(0,2) = 2

ReLU is less computationally expensive than tanh and sigmoid neurons due to its linear, non-saturating form and involving simpler mathematical operations. It's very fast to compute because its derivative is easy to handle. It is linear in the positive axis. The derivative of ReLU is:

$$\frac{\partial ReLU(x)}{\partial x} = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases} \tag{38}$$

Rectified Linear Unit (ReLU)

We glanced over the fact that ReLU is actually **not differentiable** at 0, because the left derivative (measured as moving to 0 from the left; 0) and the right derivative (measured as moving to 0 from the right; 1) are not equal. Imagine slowly moving towards zero from both sides, computing the derivative (tangent). The tangent measured on the left and right of zero are different as we reach zero. This means that the derivative of ReLU is **not defined** at 0. The left derivative of ReLU(0) is 0 and the right derivative is 1.

$$\frac{\partial ReLU(x)}{\partial x} = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \textit{not defined} & \text{if } x = 0 \end{cases} \tag{39}$$

In practice, it's relatively rare to have exactly x=0, hence, we usually don't have to worry too much about the undefined ReLU derivative when x is 0. Typically, we set it either to 0, 1, or 0.5. The dominant convention is to set it to 0, per Equation 38.

Rectified Linear Unit (ReLU)

- Advantages:
  - When $x > 0$, no saturation as in sigmoid and tanh, and so no vanishing gradient,
  - The derivative is much simpler than sigmoid and tanh and the backward pass in backpropagation is therefore faster.
- Disadvantages:
  - For ReLU activations $x < 0$ the gradient will be 0 and so there will be no learning. A neuron dies when its weights get adjusted in such a way that the weighted sum of its inputs is negative for all instances in the training set. When this happens the neuron keeps outputting zeros and the gradient of the ReLU function is zero. This is called the dying ReLU problem. In some cases a dead neuron may come back to life if weights of underlying layers are adjusted in such a way that the weighted sum of the dead neuron's inputs is positive again. This means that ReLU is not a good choice for the output layer. Why does ReLU then have such good performance? It has been shown that the dying ReLU problem regularizes the network. This means that ReLU is a good activation function when there are a lot of hidden nodes.
  - The range of ReLU is $[0, \infty]$, meaning that the activation can become very large large and cause numerical overflow.

Outline I

- A note on the technology used in this class: TensorFlow (TF)
- Start of field of Neural Networks: the Perceptron
- Continuous response variables
    - Architecture for a single continuous response
    - Architecture for multiple continuous responses
    - Loss functions for (multiple) continuous response
- Categorical response variables
    - Architecture for binary response
    - Output activation function for binary response
    - Loss function for binary response
    - Architecture for multi-label response
    - Output activation function for multi-label response
    - Loss function for multi-label response
    - Architecture for multi-class response
    - Output activation function for multi-class response
    - Loss function for multi-class response
    - Review Multi-label vs Multi-class
- Activation functions
    - What is an activation function, and do we need it?
    - What does a good activation function look like?
    - Sigmoid
    - Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

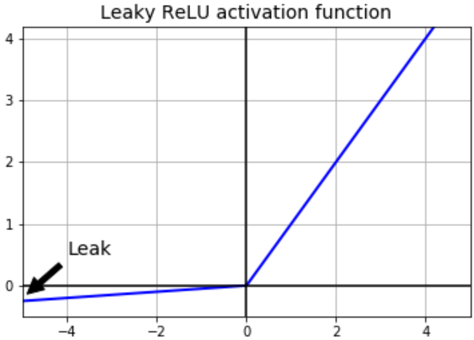
Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Leaky ReLU

$$LeakyReLU(x) = max(\alpha x, x), \tag{40}$$

with $\alpha > 0$.
Example for x=-2, α= 0.01, LeakyReLu = max(0.01(-2),-2) = -0.02
Example for x=2, α= 0.01, LeakyReLu = max(0.01(2),2) = 2



```
tf.keras.layers.Dense(n_units, activation = tf.keras.layers.LeakyReLU(alpha = 0.2))
```

Leaky ReLU

- A variant was introduced for ReLU to mitigate the *dying ReLU* problem by simply making the horizontal line into a non-horizontal component. The idea is to let the gradient be non-zero and recover during training and keep learning. Leaky units have a very small gradient instead of a zero gradient when the input is negative, giving the network a chance to continue learning.
- The hyperparameter α defines how much the function 'leaks': it is the slope of the function for $x \leq 0$ and is typically set to 0.01.

$$\frac{\partial LReLU(x)}{\partial x} = \begin{cases} \alpha & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases} \tag{41}$$

Leaky ReLU

- Advantages:
  - Fixes the dying ReLU problem.
- Disadvantages:
  - The range of Leaky ReLU is $[-\infty, \infty]$, meaning that the activation can become very small or large and cause numerical underflow/overflow.

Outline I

- A note on the technology used in this class: TensorFlow (TF)
- Start of field of Neural Networks: the Perceptron
- Continuous response variables
  - Architecture for a single continuous response
  - Architecture for multiple continuous responses
  - Loss functions for (multiple) continuous response
- Categorical response variables
  - Architecture for binary response
  - Output activation function for binary response
  - Loss function for binary response
  - Architecture for multi-label response
  - Output activation function for multi-label response
  - Loss function for multi-label response
  - Architecture for multi-class response
  - Output activation function for multi-class response
  - Loss function for multi-class response
  - Review Multi-label vs Multi-class
- Activation functions
  - What is an activation function, and do we need it?
  - What does a good activation function look like?
  - Sigmoid
  - Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Parametric ReLU (PReLU)

- Another type of ReLU that has been introduced is Parametric ReLU (PReLU). Here, instead of having α as predetermined slope like 0.01, $\alpha > 0$ is to be learned during training. It is reported that PReLU strongly outperforms ReLU on large image datasets but on smaller datasets, it runs the risk of overfitting the training set.
- The idea behind LReLU and PReLU is similar. However, Leaky ReLUs have α as a hyperparameter and Parametric ReLUs have α as a parameter.

```
tf.keras.layers.Dense(n_units, activation = tf.keras.layers.PReLU())
```

PReLU

- Advantages:
  - Fixes the dying ReLU problem.
  - Faster convergence.
- Disadvantages:
  - The range of PReLU is $[-\infty, \infty]$, meaning that the activation can become very small or large causing numerical underflow/overflow.

Outline I

- A note on the technology used in this class: TensorFlow (TF)
- Start of field of Neural Networks: the Perceptron
- Continuous response variables
  - Architecture for a single continuous response
  - Architecture for multiple continuous responses
  - Loss functions for (multiple) continuous response
- Categorical response variables
  - Architecture for binary response
  - Output activation function for binary response
  - Loss function for binary response
  - Architecture for multi-label response
  - Output activation function for multi-label response
  - Loss function for multi-label response
  - Architecture for multi-class response
  - Output activation function for multi-class response
  - Loss function for multi-class response
  - Review Multi-label vs Multi-class
- Activation functions
  - What is an activation function, and do we need it?
  - What does a good activation function look like?
  - Sigmoid
  - Tanh

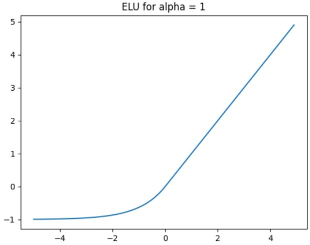
Outline II

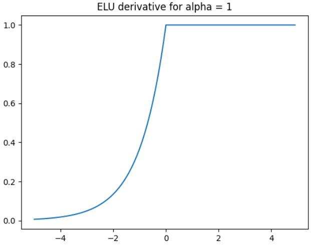
- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Exponential Linear Unit (ELU)





Exponential Linear Unit (ELU)

Similar to leaky ReLU, ELU has a small slope for negative values controlled by a hyperparameter α. Instead of a straight line, it uses a log curve. ELU outperformed all the ReLU variants in the original paper's experiments: training time was reduced and the neural network performed better on the test set. ELU is given by:

$$ELU(x) = \begin{cases} \alpha(exp(x) - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}, \tag{42}$$

where $\alpha > 0$. The partial derivative

$$\frac{\partial ELU(x)}{\partial x} = \begin{cases} \alpha exp(x) & \text{if } x \leq 0 \\ 1 & \text{if } x > 0 \end{cases}. \tag{43}$$

```
tf.keras.layers.Dense(n_units, activation = 'elu')
#or
tf.keras.layers.Dense(n_units, activation = tf.keras.layers.ELU(alpha=1.0))
```

ELU

- Advantages:
  - Fixes the dying ReLU problem, yet does have the vanishing gradient problem just like sigmoid and tanh.
- Disadvantages:
  - The range of ELU is $[-1, \infty]$, meaning that the activation can become very large, but not very small, in contrast to Leaky ReLU and PReLU
  - The main drawback of the ELU activation function is that it is slower to compute than the ReLU and its variants due to the use of the exponential function. During training this is compensated by the faster convergence rate but at test time an ELU network will be slower than a ReLU network.

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables

- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables

- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions

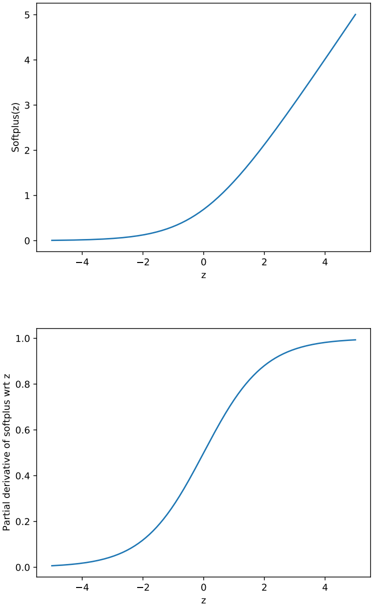
- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

Outline II

ReLU
Leaky ReLU
PReLU
ELU
Softplus
Summary

Gradient Descent
Notation
Manual implementation of Gradient Descent
Sensitivity to learning rate
Learning Rate
Gradient Descent vs Other optimizers
Gradient Descent Variants
Batch Gradient Descent
Stochastic Gradient Descent Algorithm
Mini-batch Gradient Descent Algorithm

Softplus



Softplus(z)
5
4
3
2
1
0
−4
−2
0
2
4
z
Partial derivative of softplus wrt z
1.0
0.8
0.6
0.4
0.2
0.0
−4
−2
0
2
4
z

Softplus

Softplus is a smooth approximation to ReLU:

$$Softplus(x) = ln(1 + e^x). \tag{44}$$

The partial derivative

$$\begin{aligned}
\frac{\partial Softplus(x)}{\partial x} &= \frac{1}{1+e^x}\frac{\partial(1+e^x)}{\partial x} && (45)\\
&= \frac{1}{1+e^x}(0+e^x) && (46)\\
&= \frac{e^x}{1+e^x} && (47)\\
&= \frac{\frac{e^x}{e^x}}{\frac{1}{e^x}+\frac{e^x}{e^x}} && (48)\\
&= \frac{1}{1+e^{-x}}. && \#\text{=sigmoid(x)} \quad (49)\\
& && (50)
\end{aligned}$$

```
tf.keras.layers.Dense(n_units, activation = 'softplus')
```

Softplus

- Advantages:
  - Fixes ReLU's problem of having an undefined derivative at 0
  - Fixes the dying neuron problem, yet does have the vanishing gradient problem just like sigmoid and tanh when x becomes small
- Disadvantages:
  - The range of Softplus is $[0, \infty]$, meaning that the activation can become very large (and cause numerical overflow), but not very small, in contrast to Leaky ReLU and PReLU
  - The softplus activation function is slower to compute than the ReLU and its variants due to the use of the exponential function.

Outline I

- A note on the technology used in this class: TensorFlow (TF)
- Start of field of Neural Networks: the Perceptron
- Continuous response variables
  - Architecture for a single continuous response
  - Architecture for multiple continuous responses
  - Loss functions for (multiple) continuous response
- Categorical response variables
  - Architecture for binary response
  - Output activation function for binary response
  - Loss function for binary response
  - Architecture for multi-label response
  - Output activation function for multi-label response
  - Loss function for multi-label response
  - Architecture for multi-class response
  - Output activation function for multi-class response
  - Loss function for multi-class response
  - Review Multi-label vs Multi-class
- Activation functions
  - What is an activation function, and do we need it?
  - What does a good activation function look like?
  - Sigmoid
  - Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Summary

General recommendation for hidden neuron activation:
ELU $>$ Leaky ReLU and PReLU $>$ Softplus and ReLU $>$ tanh $>$ sigmoid

Outline I

- A note on the technology used in this class: TensorFlow (TF)
- Start of field of Neural Networks: the Perceptron
- Continuous response variables
  - Architecture for a single continuous response
  - Architecture for multiple continuous responses
  - Loss functions for (multiple) continuous response
- Categorical response variables
  - Architecture for binary response
  - Output activation function for binary response
  - Loss function for binary response
  - Architecture for multi-label response
  - Output activation function for multi-label response
  - Loss function for multi-label response
  - Architecture for multi-class response
  - Output activation function for multi-class response
  - Loss function for multi-class response
  - Review Multi-label vs Multi-class
- Activation functions
  - What is an activation function, and do we need it?
  - What does a good activation function look like?
  - Sigmoid
  - Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Gradient Descent

- Gradient Descent (GD) is an iterative optimization algorithm.
- Suppose you are lost in the Great Smoky Mountains and you want to descend into the valley. This is one of these days when there is a lot of fog. It is so dense that you can't see anything. You do know that the mountain is pretty smooth, there is no danger of falling off a cliff. You can feel the slope of the ground below your feet, and you figure that to get down as quickly as possible, you just need to determine the slope in all directions, north, east, west and south, and take a step in the direction of the steepest slope.
- The example above is exactly what Gradient Descent does.

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables

- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables

- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions

- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

Outline II

ReLU
Leaky ReLU
PReLU
ELU
Softplus
Summary

Gradient Descent

Notation
Manual implementation of Gradient Descent
Sensitivity to learning rate
Learning Rate
Gradient Descent vs Other optimizers
Gradient Descent Variants
Batch Gradient Descent
Stochastic Gradient Descent Algorithm
Mini-batch Gradient Descent Algorithm

Gradient Descent - Notation

- $J(\theta)$: loss function
- $\theta \in \mathbb{R}^d$: vector of model parameters
- η: learning rate (step size)
- $\nabla_\theta J(\theta)$: gradient of the loss function with respect to the parameters

If we have one parameter, then we compute the derivative of the cost function wrt the parameter to determine in which direction (increase or decrease) we should update the parameter. If we have two or more parameters we compute partial derivatives and combine them in a vector. This vector is called the **gradient**. For example, provided two parameters (θ_0, θ_1), the gradient would look like this:

$$\nabla_\theta J(\theta) = \begin{bmatrix} \frac{\partial}{\partial \theta_0} J(\theta) \\ \frac{\partial}{\partial \theta_1} J(\theta) \end{bmatrix} \tag{51}$$

Gradient Descent - Notation

In case of the MSE as the cost function, this becomes:

$$\nabla_\theta J(\theta) = \begin{bmatrix} \frac{\partial}{\partial\theta_0} MSE(\theta) \\ \frac{\partial}{\partial\theta_1} MSE(\theta) \end{bmatrix} = \frac{2}{n} X^T (X.\theta - y) \tag{52}$$

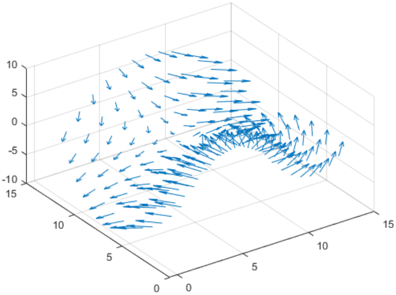
How do we use the gradient?

- GD updates parameters in the opposite direction of gradients.
- Its update equation: $\theta = \theta - \eta \nabla_\theta J(\theta)$

Let's try to visualize this. Picture a three dimensional cartesian coordinate system, with the x-axis representing θ_0, the y-axis representing θ_1 and the z-axis representing the value of the cost function. What you see is a cost surface.

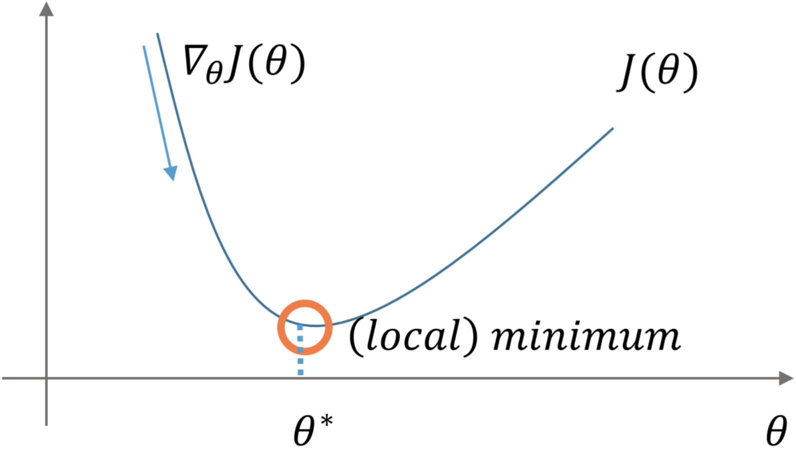
Gradient Descent

Now suppose that the surface represents a valley and we want to get to the bottom. Choose a random point in the valley (i.e., defined by θ_0 and θ_1) as starting point. Plug in the θ_0 and θ_1 in Equation 51 or 55 and multiply it by η. We get two scalars that we then subtract from the θ_0 and θ_1 that we started from using the update rule. That gives us a new point on the plot. We can then draw an arrow from the old point to the new point and that arrow is the gradient times η. That completes one update step.



Gradient Descent

Here is the same information but for one θ.



Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables

- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables

- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions

- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

Outline II

ReLU
Leaky ReLU
PReLU
ELU
Softplus
Summary

Gradient Descent

Notation
Manual implementation of Gradient Descent
Sensitivity to learning rate
Learning Rate
Gradient Descent vs Other optimizers
Gradient Descent Variants
Batch Gradient Descent
Stochastic Gradient Descent Algorithm
Mini-batch Gradient Descent Algorithm

Gradient Descent Code

A manual implementation of gradient descent is available in the file **gradient_descent.py**. The code does the following:

- Generate data for model with b=2.2222 and w=5.4675
- Sanity check: learn b and w using linear regression
- Sanity check: Exhaustive search on one parameter (w)
- Manual implementation of weight update function, 100 calls (epochs) to that function, and plotting
- Sensitivity analysis of learning rate

Results of sensitivity analysis are provided in following slides.

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables
- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables
- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions
- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

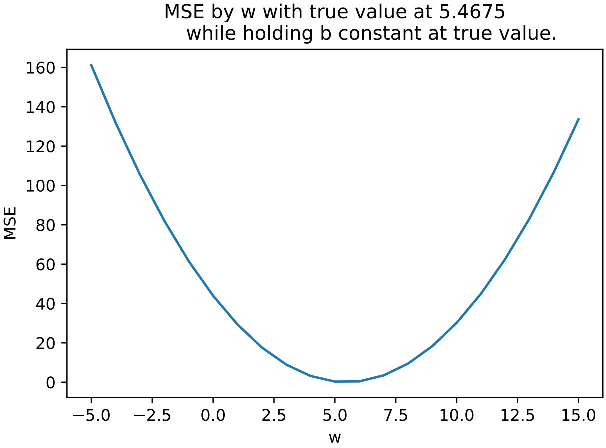
Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

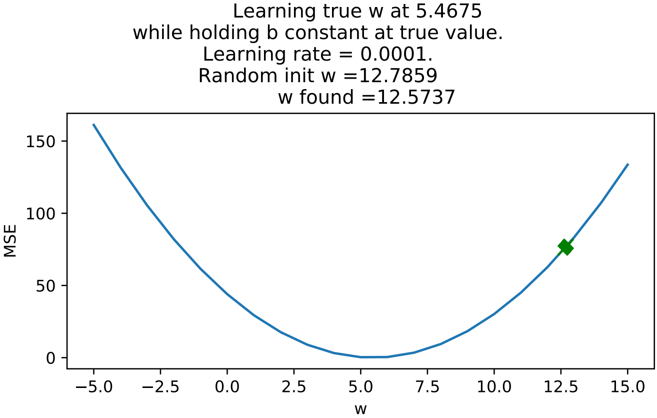
- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Gradient Descent



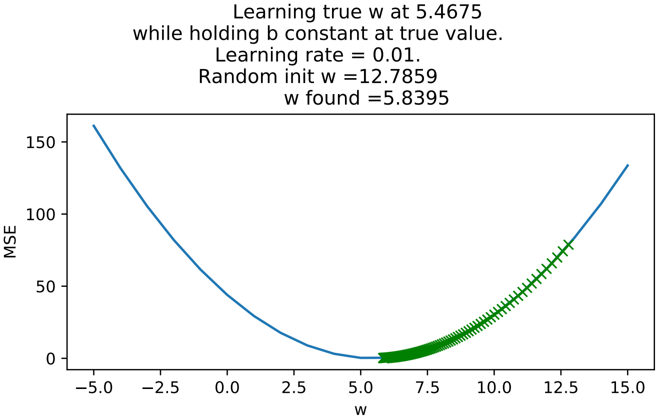
Gradient Descent

Learning rate of 0.0001. In our example, the updates are very small and since we only do 100 iterations, we won't even come close to the minimum if the randomly chosen starting value is far from the minimum.



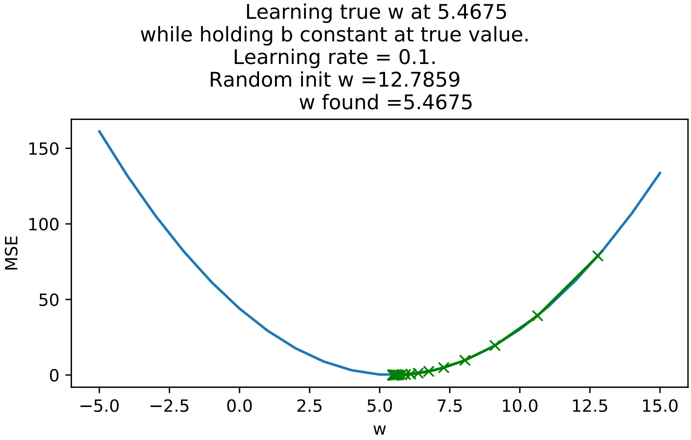
Gradient Descent

Learning rate of 0.01. We are very close, with a little more iterations we would have gotten to 5.4675.



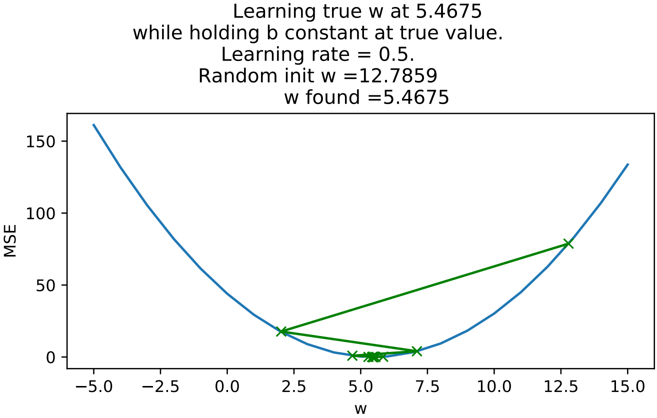
Gradient Descent

Learning rate of 0.1. It finds the optimal w right on. This is thus a good learning rate for our data.



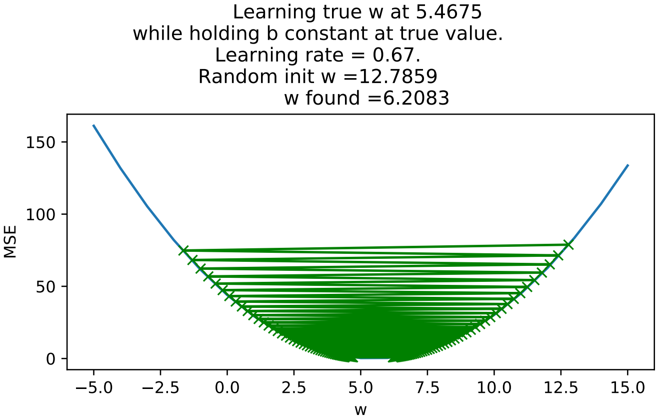
Gradient Descent

Learning rate of 0.5. We see that it starts to bounce around. Trouble will start soon. It can still find the minimum.



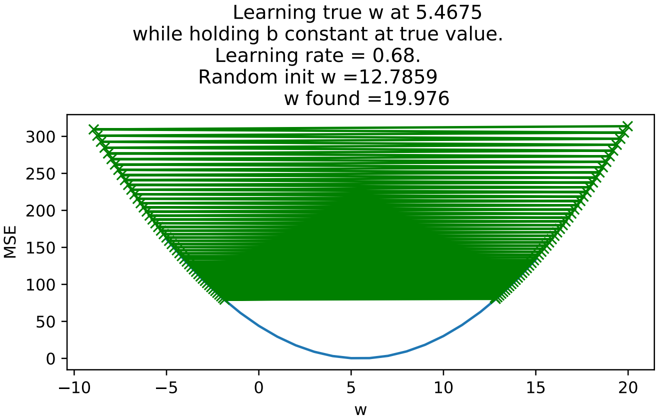
Gradient Descent

Learning rate of 0.67. This is the near the tipping point: it starts at 12.78, bounces to the other side and the MSE is nearly the same.



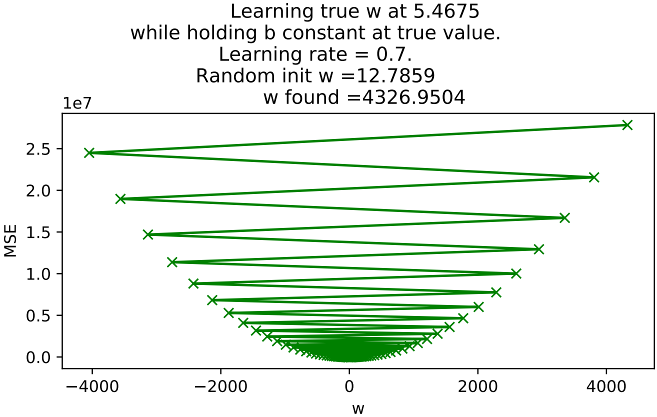
Gradient Descent

Learning rate of 0.68. This is beyond the tipping point where it no longer converges. The weight updates are now so large that weights cannot fall in between approx -1 and 12. Divergence happens, because the slope becomes increasingly steep.



Gradient Descent

Learning rate of 0.7. Faster divergence.



Outline I

- A note on the technology used in this class: TensorFlow (TF)
- Start of field of Neural Networks: the Perceptron
- Continuous response variables
  - Architecture for a single continuous response
  - Architecture for multiple continuous responses
  - Loss functions for (multiple) continuous response
- Categorical response variables
  - Architecture for binary response
  - Output activation function for binary response
  - Loss function for binary response
  - Architecture for multi-label response
  - Output activation function for multi-label response
  - Loss function for multi-label response
  - Architecture for multi-class response
  - Output activation function for multi-class response
  - Loss function for multi-class response
  - Review Multi-label vs Multi-class
- Activation functions
  - What is an activation function, and do we need it?
  - What does a good activation function look like?
  - Sigmoid
  - Tanh

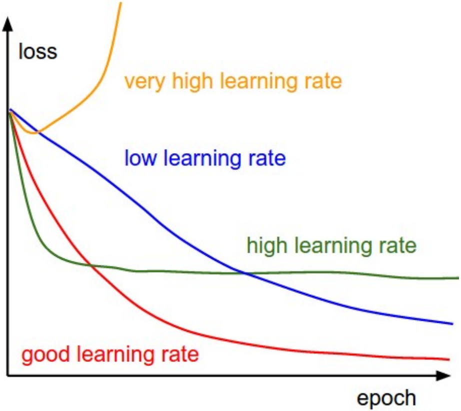
Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Learning Rate



https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables

- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables

- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions

- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

Outline II

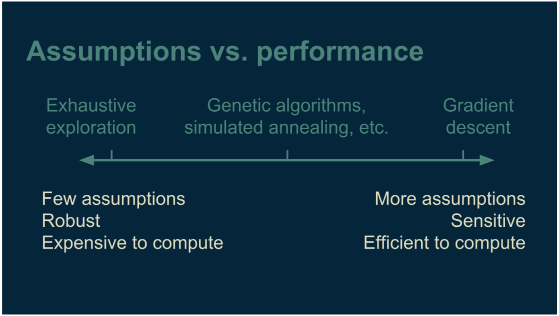
- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Gradient Descent vs Other optimizers

So what is the advantage of gradient descent over other methods? It is fast. The disadvantage is that it has assumptions, and if these assumptions are not met, then there is no guarantee that we can approximate the global minimum. The main assumption is that the loss function is convex.



Assumptions vs. performance

Exhaustive exploration

Genetic algorithms, simulated annealing, etc.

Gradient descent

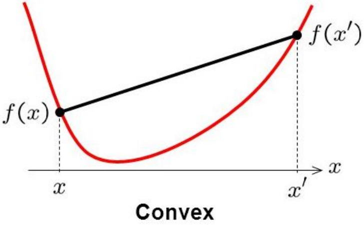
Few assumptions
Robust
Expensive to compute

More assumptions
Sensitive
Efficient to compute

https://brohrer.github.io/how_optimization_works_1.html

Gradient Descent vs Other optimizers

Convexity.



Unfortunately cost functions of neural networks (especially complex ones) are often non-convex wrt the parameters. Making Gradient Descent stochastic is a way to escape local minima.

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables

- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables

- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions

- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Gradient Descent Variants

1. Batch Gradient Descent
2. Mini-batch Gradient Descent
3. Stochastic Gradient Descent

Difference: how much data is used to compute the gradient of the objective function

Outline I

- A note on the technology used in this class: TensorFlow (TF)
- Start of field of Neural Networks: the Perceptron
- Continuous response variables
  - Architecture for a single continuous response
  - Architecture for multiple continuous responses
  - Loss functions for (multiple) continuous response
- Categorical response variables
  - Architecture for binary response
  - Output activation function for binary response
  - Loss function for binary response
  - Architecture for multi-label response
  - Output activation function for multi-label response
  - Loss function for multi-label response
  - Architecture for multi-class response
  - Output activation function for multi-class response
  - Loss function for multi-class response
  - Review Multi-label vs Multi-class
- Activation functions
  - What is an activation function, and do we need it?
  - What does a good activation function look like?
  - Sigmoid
  - Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Batch Gradient Descent

1. Computes gradient with the **entire dataset**.
2. Update equation: $\theta = \theta - \eta \nabla_{\theta} J(\theta; x, y)$
3. Pseudo code:

```
#for i in range(nb_epochs):
#  yhat = make prediction for entire dataset
#  avg_loss = compute average loss by comparing yhat and
#                    y across entire dataset
#  params_grad = compute gradient of avg_loss wrt all parameters
#  params = params - learning_rate * params_grad
```

Pros:

- Guaranteed to converge to global minimum for convex error surfaces and to a local minimum for non-convex surfaces.

Cons:

- Very slow.
- Intractable for datasets that do not fit in memory.
- No online learning

Outline I

- A note on the technology used in this class: TensorFlow (TF)
- Start of field of Neural Networks: the Perceptron
- Continuous response variables
  - Architecture for a single continuous response
  - Architecture for multiple continuous responses
  - Loss functions for (multiple) continuous response
- Categorical response variables
  - Architecture for binary response
  - Output activation function for binary response
  - Loss function for binary response
  - Architecture for multi-label response
  - Output activation function for multi-label response
  - Loss function for multi-label response
  - Architecture for multi-class response
  - Output activation function for multi-class response
  - Loss function for multi-class response
  - Review Multi-label vs Multi-class
- Activation functions
  - What is an activation function, and do we need it?
  - What does a good activation function look like?
  - Sigmoid
  - Tanh

Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

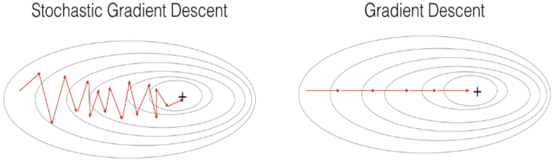
Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Stochastic Gradient Descent Algorithm

- Computes update for only one example (x_i, y_i). Randomly sampling without replacement can be beneficial.
- Update equation: $\theta = \theta - \eta \nabla_\theta J(\theta; x_i, y_i)$
- Pseudo code:

```
#for i in range(nb_epochs):
#  for ii in range(rows):
#    read next row (preferably random sample without replacement)
#    yhat = make prediction for single row
#    loss = compute loss by comparing yhat and y for single row
#    params_grad = compute gradient of loss wrt all parameters
#    params = params - learning_rate * params_grad
```



Stochastic Gradient Descent Algorithm

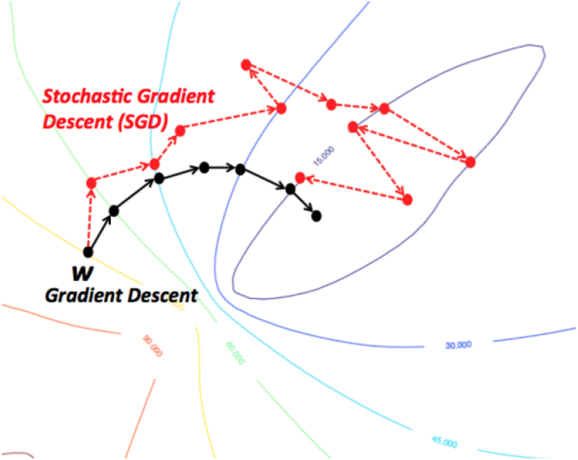
Pros:

- Much faster than batch gradient descent.
- Allows online learning
- Can jump out of local minima

Cons:

- SGD performs frequent updates with a high variance

Stochastic Gradient Descent Algorithm



SGD shows same convergence behavior as batch gradient descent if learning rate is slowly decreased (annealed) over time

Outline I

A note on the technology used in this class: TensorFlow (TF)

Start of field of Neural Networks: the Perceptron

Continuous response variables

- Architecture for a single continuous response
- Architecture for multiple continuous responses
- Loss functions for (multiple) continuous response

Categorical response variables

- Architecture for binary response
- Output activation function for binary response
- Loss function for binary response
- Architecture for multi-label response
- Output activation function for multi-label response
- Loss function for multi-label response
- Architecture for multi-class response
- Output activation function for multi-class response
- Loss function for multi-class response
- Review Multi-label vs Multi-class

Activation functions

- What is an activation function, and do we need it?
- What does a good activation function look like?
- Sigmoid
- Tanh

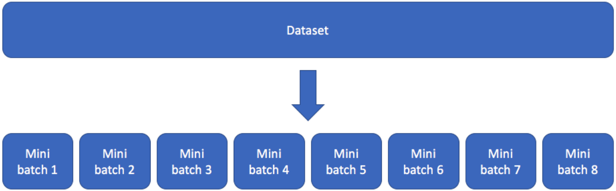
Outline II

- ReLU
- Leaky ReLU
- PReLU
- ELU
- Softplus
- Summary

Gradient Descent

- Notation
- Manual implementation of Gradient Descent
- Sensitivity to learning rate
- Learning Rate
- Gradient Descent vs Other optimizers
- Gradient Descent Variants
- Batch Gradient Descent
- Stochastic Gradient Descent Algorithm
- Mini-batch Gradient Descent Algorithm

Mini-batch Gradient Descent Algorithm



Dataset

Mini batch 1

Mini batch 2

Mini batch 3

Mini batch 4

Mini batch 5

Mini batch 6

Mini batch 7

Mini batch 8

- Mini-batch gradient descent takes the best of both worlds and performs an update for every mini-batch of n examples.
- Update equation: $\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x_{(i:i+n)}; y_{(i:i+n)})$

```
#for i in range(nb_epochs):
# for ii in range(batches):
#   read next batch of rows (prefer. rand. sample w/o replace.)
#   yhat = make prediction for batch of rows
#   avg_loss = compute loss by comparing yhat and y for batch
#   params_grad = compute gradient of avg_loss wrt all parameters
#   params = params - learning_rate * params_grad
```

Mini-batch Gradient Descent Algorithm

Pros:

- reduces the variance of the parameter updates, which can lead to more stable convergence.

Cons:

- Mini-batch size is a hyperparameter. Common sizes are 50-250 but can vary for different applications.

Note that mini-batch GD is

- Typically the algorithm of choice,
- Usually referred to as SGD even when mini-batches are used.

Comparison of trade-offs of gradient descent variants

Method	**Accuracy**	**Update Speed**	**Memory Usage**	**Online Learning**
Batch gradient descent	Good	Slow	High	No
Stochastic gradient descent	Good (with annealing)	High	Low	Yes
Mini-batch gradient descent	Good	Medium	Medium	Yes

Details on how batch and mini-batch gradient descent update the weights

- In the previous few slides we said that we compute the loss for each of the instances in the batch and then average those losses across the batch. Next we said that we compute the gradient of this average loss with regard to each weight and use the update rule (i.e., we use gradient descent).
- Formally the
$$\frac{\partial L}{\partial w_j} = \frac{\partial}{\partial w_j}\frac{1}{n}\sum_{i=1}^{n} L_i \tag{53}$$
where j indicates a specific weight, and n indicates the number of instances in the batch.
- Alternatively, since $(f+g)' = f' + g'$, we can rewrite Equation 54 as
$$\frac{\partial L}{\partial w_j} = \frac{1}{n}\sum_{i=1}^{n}\frac{\partial L_i}{\partial w_j} \tag{54}$$
In other words we are computing the average of the gradients, and we can then use that in gradient descent just like Equation 54. The result is equivalent.

Vectorizing the update across multiple weights (and/or instances)

Example with MSE.

Consider the gradient vector for the MSE loss function

$$\nabla_\theta MSE(\theta) = \begin{bmatrix} \frac{\partial}{\partial\theta_0} MSE(\theta) \\ \frac{\partial}{\partial\theta_1} MSE(\theta) \\ \vdots \\ \frac{\partial}{\partial\theta_m} MSE(\theta) \end{bmatrix} = \frac{2}{n} X^T (X.\theta - y) \tag{55}$$

where m denotes the number of weights, θ and y are each column vectors, and X is a matrix with instances in the rows and weights in the columns. In the update rule

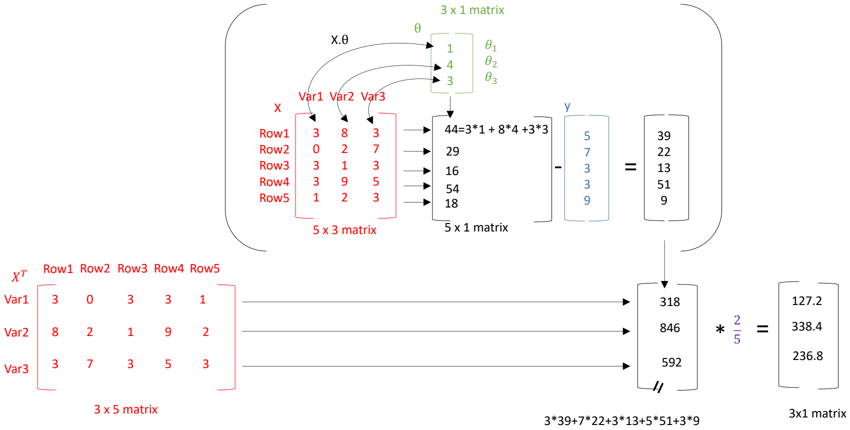
$$\theta = \theta - \eta \nabla_\theta MSE(\theta) \tag{56}$$

θ is a column vector of weights. On the next slide we will work out the matrix multiplication revealing how the update is vectorized (meaning we do not need to loop through each weight).

Vectorizing the update across multiple weights (and/or instances)

Example with MSE.

$$\nabla_\theta \text{MSE}(\theta) = \frac{2}{n} X^T \, (X.\theta \; - \text{y})$$



3 x 1 matrix

X.θ

$$\theta = \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix} \begin{matrix} \theta_1 \\ \theta_2 \\ \theta_3 \end{matrix}$$

$$X = \begin{array}{c|ccc} & \text{Var1} & \text{Var2} & \text{Var3} \\ \hline \text{Row1} & 3 & 8 & 3 \\ \text{Row2} & 0 & 2 & 7 \\ \text{Row3} & 3 & 1 & 3 \\ \text{Row4} & 3 & 9 & 5 \\ \text{Row5} & 1 & 2 & 3 \end{array}$$

5 x 3 matrix

$$\begin{bmatrix} 44=3*1 + 8*4 +3*3 \\ 29 \\ 16 \\ 54 \\ 18 \end{bmatrix} - \overset{\text{y}}{\begin{bmatrix} 5 \\ 7 \\ 3 \\ 3 \\ 9 \end{bmatrix}} = \begin{bmatrix} 39 \\ 22 \\ 13 \\ 51 \\ 9 \end{bmatrix}$$

5 x 1 matrix

$$X^T = \begin{array}{c|ccccc} & \text{Row1} & \text{Row2} & \text{Row3} & \text{Row4} & \text{Row5} \\ \hline \text{Var1} & 3 & 0 & 3 & 3 & 1 \\ \text{Var2} & 8 & 2 & 1 & 9 & 2 \\ \text{Var3} & 3 & 7 & 3 & 5 & 3 \end{array}$$

3 x 5 matrix

$$\begin{bmatrix} 318 \\ 846 \\ 592 \end{bmatrix} * \frac{2}{5} = \begin{bmatrix} 127.2 \\ 338.4 \\ 236.8 \end{bmatrix}$$

3*39+7*22+3*13+5*51+3*9

3x1 matrix

Vectorizing the update across multiple weights (and/or instances)

Example with MSE.

$$\theta = \theta - \eta \nabla_{\theta} \mathrm{MSE}(\theta)$$

$$\begin{array}{ccccccc} \theta & & \theta & & \eta & & \nabla_{\theta}\mathrm{MSE}(\theta) \\ \begin{bmatrix} -0.272 \\ 0.616 \\ 0.632 \end{bmatrix} & = & \begin{bmatrix} 1 \\ 4 \\ 3 \end{bmatrix} & - & 0.01 & * & \begin{bmatrix} 127.2 \\ 338.4 \\ 236.8 \end{bmatrix} \\ \text{3x1 matrix} & & \text{3 x 1 matrix} & & & & \text{3x1 matrix} \end{array}$$

Notation clarification

- In the code or slides you will sometimes see that we compute the partial derivative (i.e., one weight at a time) as $\frac{\partial MSE}{\partial \theta_j} = \frac{2}{n}\sum_{i=1}^{n}(X_i\theta - y_i)X_{i,j}$, which is equivalent to $\frac{\partial MSE}{\partial \theta_j} = \frac{2}{n}\sum_{i=1}^{n} X_{i,j}(X_i\theta - y_i)$
- When we are dealing with multiple weights we need to think carefully about matrix multiplication, the dimensions involved, and the position of the product terms. You will see that we compute the gradient (i.e., collection of partial derivatives) as $\nabla_\theta MSE(\theta) = \frac{2}{n}X^T(X.\theta - y)$, which is equivalent to $\nabla_\theta MSE(\theta) = \frac{2}{n}(\theta^T X^T - y^T)X$.
- Also note that $\frac{\partial MSE}{\partial \theta_j} = \frac{2}{n}\sum_{i=1}^{n}(X_i\theta - y_i)X_{i,j} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - X_i\theta)X_{i,j}$