Gradient Descent

Towards Neural Networks

Justin Stevens
Undergraduate AI Society
April 2nd, 2019

Outline

1. Decision Making
   - Perceptrons
   - Activation Functions
2. Classifying Digits through MNIST
3. Improvements

Should I Stay or Should I Go?

Let's say I'm deciding on a given day whether or not to go to an Edmonton Oilers game. Here are the factors that will influence my decision:

- Are the tickets cheap or expensive?
- Do I have the time to go?
- Do I care about the team they're playing?

We'll make my decision by encoding each possible input as a vector $\bar{\mathbf{x}}$:

Ticket Prices	Availability	Interest	$\bar{\mathbf{x}}$
Cheap	Yes	Yes	$(1, 1, 1)$
Cheap	No	No	$(1, 0, 0)$
Cheap	Yes	No	$(1, 1, 0)$
Cheap	No	Yes	$(1, 0, 1)$
Expensive	Yes	Yes	$(0, 1, 1)$
Expensive	No	No	$(0, 0, 0)$
Expensive	No	Yes	$(0, 0, 1)$
Expensive	Yes	No	$(0, 1, 0)$

How Will I Make my Decision?

Let's say I don't care much about price, but I do care about my availability and interest. In this case, the corresponding weights might be $\bar{\mathbf{w}} = (1, 6, 3)$. We can then compute the dot product $\bar{\mathbf{w}} \cdot \bar{\mathbf{x}}$ for each possible input:

Ticket Prices	Availability	Interest	$\bar{\mathbf{x}}$	$\bar{\mathbf{w}} \cdot \bar{\mathbf{x}}$
Cheap	Yes	Yes	$(1, 1, 1)$	10
Cheap	No	No	$(1, 0, 0)$	1
Cheap	Yes	No	$(1, 1, 0)$	7
Cheap	No	Yes	$(1, 0, 1)$	4
Expensive	Yes	Yes	$(0, 1, 1)$	9
Expensive	No	No	$(0, 0, 0)$	0
Expensive	No	Yes	$(0, 0, 1)$	3
Expensive	Yes	No	$(0, 1, 0)$	6

We can now define my **activation threshold**, t, which will determine whether or not I go to the game, represented in binary.

Formula for Decision Making

The general formula for my decision to go to the Oilers game is

$$\text{output} = \begin{cases} 0 & \text{if } \bar{\mathbf{w}} \cdot \bar{\mathbf{x}} < t \\ 1 & \text{if } \bar{\mathbf{w}} \cdot \bar{\mathbf{x}} \geq t. \end{cases}$$

For instance, if $t = 9$, we see I'll only go if I'm both available and interested. If $t = 7$, I'll also go if the tickets are cheap and I'm available:

Ticket Prices	Availability	Interest	$\bar{\mathbf{x}}$	$\bar{\mathbf{x}} \cdot \bar{\mathbf{w}}$
Cheap	**Yes**	**Yes**	$(1,1,1)$	**10**
Cheap	No	No	$(1,0,0)$	1
Cheap	**Yes**	**No**	$(1,1,0)$	**7**
Cheap	No	Yes	$(1,0,1)$	4
Expensive	**Yes**	**Yes**	$(0,1,1)$	**9**
Expensive	No	No	$(0,0,0)$	0
Expensive	No	Yes	$(0,0,1)$	3
Expensive	Yes	No	$(0,1,0)$	6

This is a simplified model of a **perceptron**. The idea was developed by Frank Rosenblatt at Cornell in 1957, and is often used in psychology.

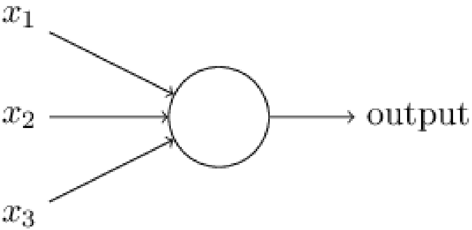

Figure 1: *Source: Nielsen*

Each of these lines collect evidence and are weighted to produce an output. In practice, our inputs and outputs don't necessarily have to be binary; they can be real-valued. We therefore have to define a new activation function.

Introducing the Bias

Instead of comparing our weighted sum to a threshold, we instead *add* a bias, b, to our weighted sum. We write this as $\bar{\mathbf{w}} \cdot \bar{\mathbf{x}} + b$ instead. Then

$$\text{output} = \begin{cases} 0 & \text{if } \bar{\mathbf{w}} \cdot \bar{\mathbf{x}} + b < 0 \\ 1 & \text{if } \bar{\mathbf{w}} \cdot \bar{\mathbf{x}} + b \geq 0. \end{cases}$$

This is known as the *heaviside step function*. We'll extend our model to multiple outputs soon, but first we'll examine other activation functions.

Rectified Linear Unit

If we want our outputs to be non-negative, we use the **rectified linear unit**,

$$f(x) = \max\{0, x\}.$$

Graphically, we can see:

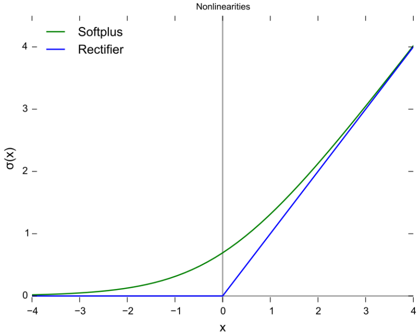

Figure 2: Rectifier, and a smooth approximation $\log(1 + e^x)$. (*Source: Wikipedia*).

Sigmoid Function

As we saw above, our output doesn't necessarily have to be a 0 or 1; using a rectified linear unit, it can be any non-negative number. However, for computational purposes, it's easiest if our outputs live in the range $(0, 1)$. We now define the **sigmoid** or logistic function, $\sigma(z) = \frac{1}{1+e^{-z}}$. Graphically,

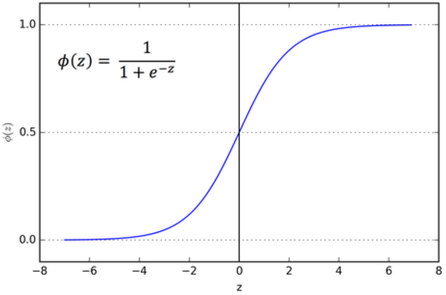

Figure 3: As $z \to \infty$, we see $\sigma(z) \to 1$. Alternatively, as $z \to -\infty$, $\sigma(z) \to 0$. (*Source: Towards Data Science*).

Outline

Example Images

In **supervised learning** problems, we're given a set of training data with labels, which we try to learn. We'll use a generalization of the perceptron with different neurons, for which we try to learn the best possible weights.

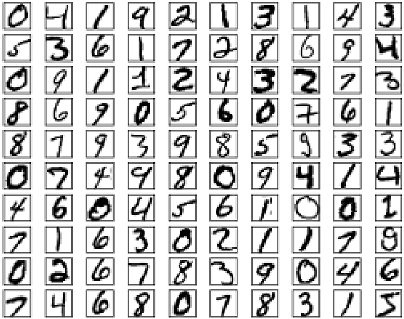

Figure 4: How would you devise a system for a **computer** to classify the digits? How can we best utilize the data set, known as MNIST? (*Source: Nielsen*)

MNIST Dataset

- The MNIST database contains seventy thousand handwritten digits.
  - Each data-point contains both an image, and the desired digit.
  - $60,000$ images are designated for training, and $10,000$ for testing:

```python
import tensorflow as tf
from tensorflow import keras
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
```

- Each image contains pixels ranging 0 to 255, in decreasing darkness.
- An individual image is a 28×28 array of pixels.
- The desired digit is represented as a number from 0 to 9.

We'll build a model from the training images that will learn to classify digits!

What we're building towards

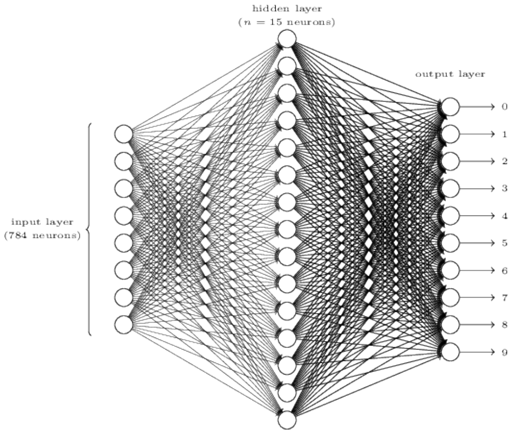

Figure 5: A simple neural network structure. The input vectors on the left hand side have $28 \times 28 = 784$ inputs for each pixel, and the output layer has 10 digits. (*Source: Nielsen*)

Extending our Model

All of our weights and bias will be initialized from a normal distribution with mean 0 and standard deviation 1.

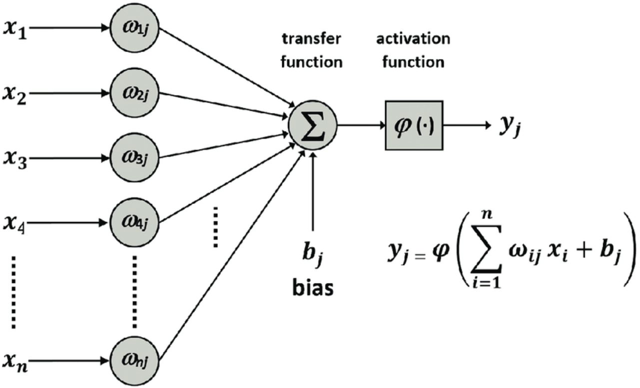

Figure 6: *Source: Daniel Alvarez, InTech*

Hidden Layer

The role of the **hidden layer** is to hold intermediate calculations. These will in turn be used to compute the output layer. To produce the hidden layer, we must have an 784×15 weight matrix, as seen below:

$$W = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1,784} \\ w_{21} & w_{22} & \cdots & w_{2,784} \\ \vdots & \vdots & \ddots & \vdots \\ w_{15,1} & w_{15,2} & \cdots & w_{15,784} \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_{784} \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{15} \end{pmatrix}.$$

We take the dot product of each **row** with our input vector $\mathbf{x}$. We then add our bias vector, $\mathbf{b}$, which is 15×1. We finally apply our activation:

$$\mathbf{h} = \sigma(W\mathbf{x} + \mathbf{b}).$$

Notice the sigmoid function is applied component wise.

Output Layer

We must now define a transformation from $\mathbb{R}^{15}$ to $\mathbb{R}^{10}$, which we can do using a 10×15 weight matrix $\hat{W}$. We can then add a 10×1 bias vector, $\hat{\mathbf{b}}$.

We can write this as $\mathbf{f} = \begin{pmatrix} \hat{w}_{11} & \hat{w}_{12} & \cdots & \hat{w}_{1,15} \\ \hat{w}_{21} & \hat{w}_{22} & \cdots & \hat{w}_{2,15} \\ \vdots & \vdots & \vdots & \vdots \\ \hat{w}_{10,1} & \hat{w}_{10,2} & \cdots & \hat{w}_{10,15} \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \\ \vdots \\ \vdots \\ h_{15} \end{pmatrix} + \begin{pmatrix} \hat{b}_1 \\ \hat{b}_2 \\ \vdots \\ \hat{b}_{10} \end{pmatrix}$.

We aren't done yet! We want the output to be the probability an image is a specific digit. To do so, we use a **softmax** activation. The formula is

$$\text{softmax}(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{10} e^{z_k}}, \quad 1 \leq j \leq 10.$$

Notice the sum of these values will always be 1. The full computation is

$$\mathbf{f} = \hat{W}\mathbf{h} + \hat{\mathbf{b}}, \quad \mathbf{o} = \text{softmax}(\mathbf{f}).$$

One Hot Encoding

Once we've computed the output, we need a way to compare it to our desired result. However, $\mathbf{o}$ is a 10×1 vector, whereas our desired digit $y_{\text{train}}(\mathbf{x})$ is a scalar. We therefore encode the digit as a 10×1 vector:

$$
\begin{array}{ccccc}
\begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} &
\begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} &
\begin{pmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix} &
\cdots &
\begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix} \\
\hline
0 & 1 & 2 & \cdots & 9
\end{array}
$$

The code for this is relatively simple:

```
y_test=keras.utils.to_categorical(y_test, num_classes=10)
y_train=keras.utils.to_categorical(y_train, num_classes=10)
```

Negative Log Likelihood

To compute how accurate our model was at predicting a given value, we need a **loss** function. In this case, it's easiest to use *negative log likelihood*.

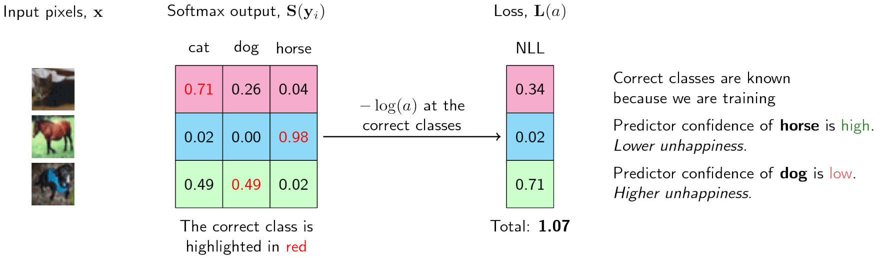

Figure 7: *Source: LJ Mirand*

To compute the loss for an individual training example, $\mathbf{x}$, with one-hot encoded label $y_{\text{train}}(\mathbf{x})$, and output $\mathbf{o}$, we compute

$$L(\mathbf{x}) = -y_{\text{train}}(\mathbf{x}) \cdot \log \mathbf{o} = -\log(o_j),$$

where j is the true label.

Graph of Negative Log

Recall $L(\mathbf{x}) = -\log(o_j)$. Since o_j is between 0 and 1, we can graph the function, noting it approaches 0 as $o_j \to 1$, and goes to ∞ as $o_j \to 0$.

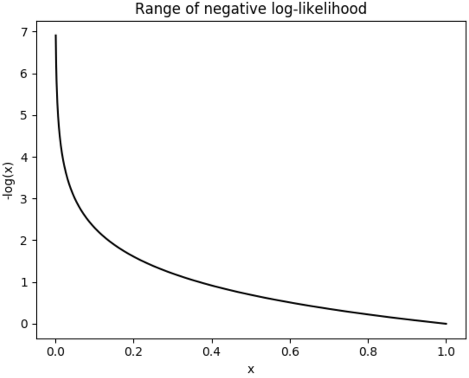

Figure 8: *Source: LJ Mirand*

Summarizing the Loss Function

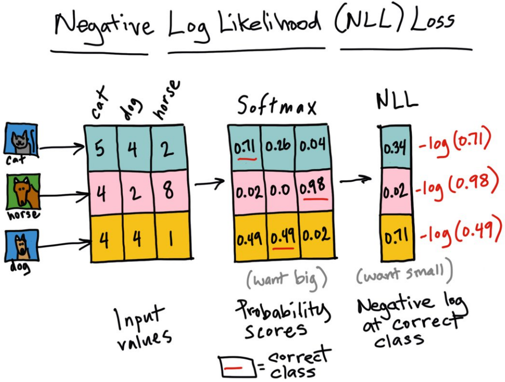

Figure 9: *Source: Micheleen Harris*

Gradient Descent Intuition

Our goal is to minimize loss with respect to weights and biases. To simplify the model, let's assume C is a function of two variables: v_1 and v_2.

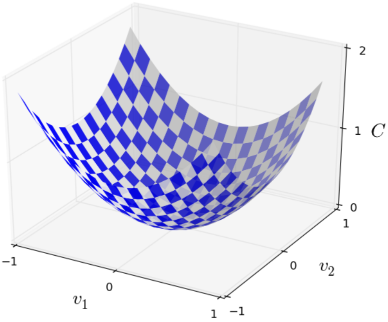

Figure 10: *Source: Nielsen*

Gradient Descent II

We see how much C changes as we nudge v_1 and v_2. Approximating,

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2.$$

We find the optimal way to adjust Δv_1 and Δv_2 so that ΔC is negative. We define $\Delta v = \binom{\Delta v_1}{\Delta v_2}$ and the **gradient** vector $\nabla C = \begin{pmatrix} \frac{\partial C}{\partial v_1} & \frac{\partial C}{\partial v_2} \end{pmatrix}^T$. Therefore,

$$\Delta C \approx \nabla C \cdot \Delta v.$$

By the Cauchy-Schwartz inequality, the direction of greatest descent is $\Delta v = -\lambda \nabla C$. In this case, $\Delta C \approx \nabla C \cdot -\lambda \nabla C = -\lambda \left\| \nabla C \right\|^2 \ll 0$.

We therefore update $v \to v' = v - \lambda \nabla C$, where λ is the **learning rate**. We iterate this process in the hopes of finding a local minimum of C. This process can fairly easily generalize from 2 variables to n variables if we wish.

Gradient Descent III

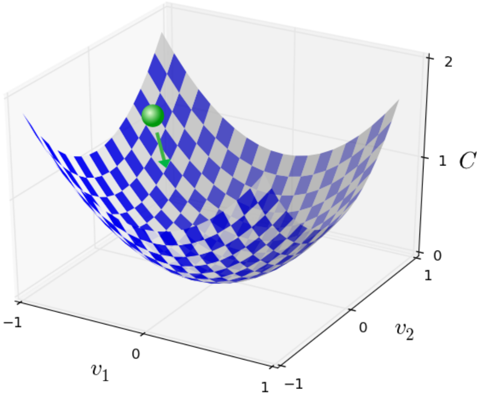

Figure 11: *Source: Nielsen*

Gradient Descent IV

Our general rule for updating weights and biases in our neural network is

$$w_k \to w_k' = w_k - \eta \frac{\partial L}{\partial w_k}, \quad b_\ell \to b_\ell' = b_\ell - \eta \frac{\partial L}{\partial b_\ell}.$$

In practice, updating our weights at every step is subject to stochastic error. Therefore, we approximate our gradient function (which holds all of the partial derivatives) in **batches** of size m and take their average:

$$\nabla L \approx \frac{1}{m} \sum_{i=1}^{m} \nabla L_{X_i}.$$

We can rewrite our update steps as $\begin{cases} w_k \to & w_k' = w_k - \frac{\eta}{m} \sum_{i=1}^{m} \frac{\partial L_{X_i}}{\partial w_k} \\ b_\ell \to & b_\ell' = b_\ell - \frac{\eta}{m} \sum_{i=1}^{m} \frac{\partial L_{X_i}}{\partial w_k} \end{cases}.$

Backpropagation Part I

In our hidden layer, we have $784 \times 15 + 15 \times 1 = 11,775$ weights and biases to train on. In the output layer, we only have $15 \times 10 + 10 \times 1 = 160$. We'll answer the question: how much does our loss function depend on these parameters? To answer this, we need the chain rule from calculus:

$$L = -\log(o_j), \quad o_j = \frac{e^{f_j}}{e^{f_1} + e^{f_2} + \cdots + e^{f_{10}}}.$$

Then $\frac{\partial L}{\partial f_j} = \frac{\partial L}{\partial o_j} \frac{\partial o_j}{\partial f_j} = -\frac{1}{o_j} \frac{\partial o_j}{\partial f_j}$. Using the quotient rule and some algebra,

$$\begin{aligned}
\frac{\partial L}{\partial f_j} &= -\frac{1}{o_j} \frac{\left(e^{f_1} + \cdots + e^{f_{10}}\right) e^{f_j} - e^{f_j} e^{f_j}}{\left(e^{f_1} + \cdots + e^{f_{10}}\right)^2} \\
&= -\frac{e^{f_1} + \cdots + e^{f_{10}}}{e^{f_j}} \cdot \frac{e^{f_j}\left(e^{f_1} + e^{f_2} + \cdots + e^{f_{10}} - e^{f_j}\right)}{\left(e^{f_1} + \cdots + e^{f_{10}}\right)^2} \\
&= -\frac{e^{f_1} + \cdots + e^{f_{10}} - e^{f_j}}{e^{f_1} + \cdots + e^{f_{10}}} = -(1 - o_j) = o_j - 1.
\end{aligned}$$

Backpropagation Part II

$$L = -\log\left(o_j\right), \quad o_j = \frac{e^{f_j}}{e^{f_1} + e^{f_2} + \cdots + e^{f_{10}}}.$$

If $i \neq j$, by the chain rule $\frac{\partial L}{\partial f_i} = \frac{\partial L}{\partial o_j}\frac{\partial o_j}{\partial f_i} = -\frac{1}{o_j}\frac{\partial o_j}{\partial f_i}$. Using the quotient rule,

$$\begin{aligned}
\frac{\partial L}{\partial f_i} &= -\frac{1}{o_j} \cdot \frac{-e^{f_j} e^{f_i}}{\left(e^{f_1} + e^{f_2} + \cdots + e^{f_{10}}\right)^2} \\
&= \frac{e^{f_1} + e^{f_2} + \cdots + e^{f_{10}}}{e^{f_j}} \cdot \frac{e^{f_j} e^{f_i}}{\left(e^{f_1} + e^{f_2} + \cdots + e^{f_{10}}\right)^2} \\
&= \frac{e^{f_i}}{e^{f_1} + \cdots + e^{f_{10}}} = o_i.
\end{aligned}$$

We've now computed how much our loss function depends upon the final activations before applying the softmax. We now go back a layer and see how much L depends upon $\hat{W}$ and $\hat{b}$, and store these results in a gradient.

Backpropagation Part III

We can write this as $\mathbf{f} = \begin{pmatrix} \hat{w}_{11} & \hat{w}_{12} & \cdots & \hat{w}_{1,15} \\ \hat{w}_{21} & \hat{w}_{22} & \cdots & \hat{w}_{2,15} \\ \vdots & \vdots & \vdots & \vdots \\ \hat{w}_{10,1} & \hat{w}_{10,2} & \cdots & \hat{w}_{10,15} \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \\ \vdots \\ \vdots \\ h_{15} \end{pmatrix} + \begin{pmatrix} \hat{b}_1 \\ \hat{b}_2 \\ \vdots \\ \hat{b}_{10} \end{pmatrix}$.

By matrix multiplication, $f_i = \hat{w}_{i1}h_1 + \hat{w}_{i2}h_2 + \cdots + \hat{w}_{i,15}h_{15} + \hat{b}_i$. Therefore, $\frac{\partial L}{\partial \hat{w}_{k\ell}} = \sum_{i=i}^{10} \frac{\partial L}{\partial f_i}\frac{\partial f_i}{\partial \hat{w}_{k\ell}} = \frac{\partial L}{\partial f_k}\frac{\partial f_k}{\partial \hat{w}_{k\ell}} = \begin{cases} o_k h_\ell & \text{if } k \neq j \\ (o_k - 1)\, h_\ell & \text{if } k = j \end{cases}$. We can similarly see $\frac{\partial L}{\partial \hat{b}_k} = \sum_{i=1}^{10} \frac{\partial L}{\partial f_i}\frac{\partial f_i}{\partial \hat{b}_k} = \frac{\partial L}{\partial f_k}\frac{\partial f_k}{\partial \hat{b}_k} = \begin{cases} o_k & \text{if } k \neq j \\ o_k - 1 & \text{if } k = j. \end{cases}$.

Finally, we can observe $\frac{\partial f_i}{\partial h_k} = \hat{w}_{ik}$ by the way we set up the matrix. We store all of these values in a gradient table for easy access later. This is a core feature of **dynamic programming**, which is essential in this program.

Backpropagation IV

Recall in the hidden layer, $h = \sigma(W\mathbf{x} + \mathbf{b})$, where

$$W = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1,784} \\ w_{21} & w_{22} & \cdots & w_{2,784} \\ \vdots & \vdots & \ddots & \vdots \\ w_{15,1} & w_{15,2} & \cdots & w_{15,784} \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_{784} \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{15} \end{pmatrix}.$$

By matrix multiplication, $h_k = \sigma(w_{k1}x_1 + w_{k2}x_2 + \cdots + w_{k,784}x_{784} + b_k)$. By the chain rule, we note $w_{k\ell}$ is fed into every final layer, therefore

$$\begin{aligned} \frac{\partial L}{\partial w_{k\ell}} &= \sum_{i=1}^{10} \frac{\partial L}{\partial f_i} \frac{\partial f_i}{\partial h_k} \frac{\partial h_k}{\partial w_{k\ell}} \\ &= \left(\sum_{i \neq j} o_i w_{ik} + (o_j - 1) w_{jk} \right) \frac{\partial h_k}{\partial w_{k\ell}}. \end{aligned}$$

Backpropagation Finale

Let $g_k = w_{k1}x_1 + w_{k2}x_2 + \cdots + w_{k,784}x_{784} + b_k$. Then $h_k = \sigma(g_k)$ hence

$$\frac{\partial h_k}{\partial w_{k\ell}} = \frac{\partial h_k}{\partial g_k}\frac{\partial g_k}{\partial w_{k\ell}}.$$

Recall $\sigma(z) = \frac{1}{1+e^{-z}}$, so $\frac{d}{dz}\sigma(z) = \frac{e^{-z}}{1+e^{-z}} \cdot \frac{1}{1+e^{-z}} = (1-\sigma(z))\,\sigma(z)$. Then

$$\frac{\partial h_k}{\partial w_{k\ell}} = (1-\sigma(g_k))\,\sigma(g_k)\mathbf{x}_\ell.$$

Similarly, we see $\frac{\partial h_k}{\partial b_k} = (1-\sigma(g_k))\,\sigma(g_k)$, since $\frac{\partial g_k}{\partial b_k} = 1$. Thus,

$$\frac{\partial L}{\partial w_{k\ell}} = \left(\sum_{i\neq j} o_i w_{ik} + (o_j - 1)w_{jk}\right)(1-\sigma(g_k))\,\sigma(g_k)\mathbf{x}_\ell$$

$$\frac{\partial L}{\partial b_k} = \left(\sum_{i\neq j} o_i w_{ik} + (o_j - 1)w_{jk}\right)(1-\sigma(g_k))\,\sigma(g_k).$$

Epoch

- For each batch of size m training images,
  - For each i in range $\{1, 2, 3, \cdots, m\}$
    - Randomly initialized a 784×15 weight matrix W, a 15×10 matrix $\hat{W}$, a 15×1 bias vector b, and a 10×1 vector $\hat{b}$.
    - Performed feed forward on $\mathbf{x}_i$ to compute $\mathbf{o}, \mathbf{h}$, and $\mathbf{f}$. Saved results.
    - Compared our predicted result with the actual result to compute a loss using negative log likelihood: $L(\mathbf{x_i}) = -\log(o_j)$, where j is the true label.
    - Performed back propagation to compute gradient and see how much our loss depended upon every weight and bias.
  - Average gradients for the batch, and apply gradient descent to each component with **fixed** learning rate η.

This process is one epoch. It's often helpful to pass our data through our training model multiple times to get the best weights and biases.

If we wish to use a **varied** learning rate, we can use a different optimizer, such as *stochastic gradient descent*. We can then use "sgd", or an even more optimized version such as "adam". For more info, see here.

Outline

Overfitting

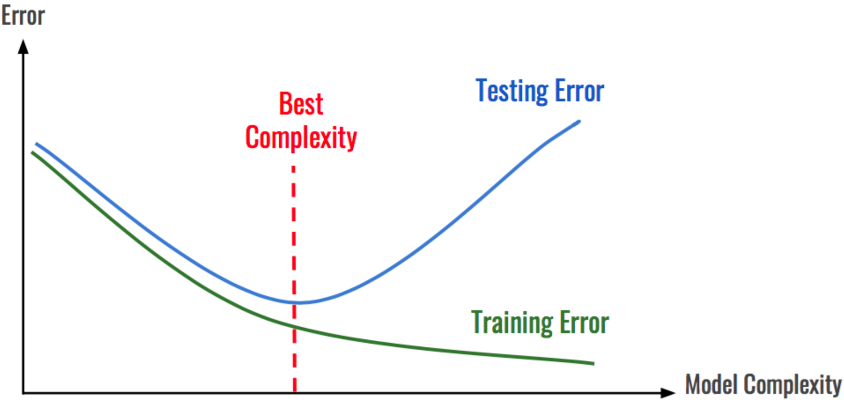

Figure 12: *Source: Hacker Noon*

References

- Michael Nielsen: Using neural nets to recognize handwritten digits
- LJ Mirand: Understanding softmax and the negative log-likelihood
- Towards Data Science: A Beginner's Guide to Neural Networks
- Deep Learning, Goodfellow et. al.
- 3d Visualizing a Neural Network
- Visualizing Gradient Descent
- Tensorflow Tutorials