Neural Network Crash Course

1 A single neuron

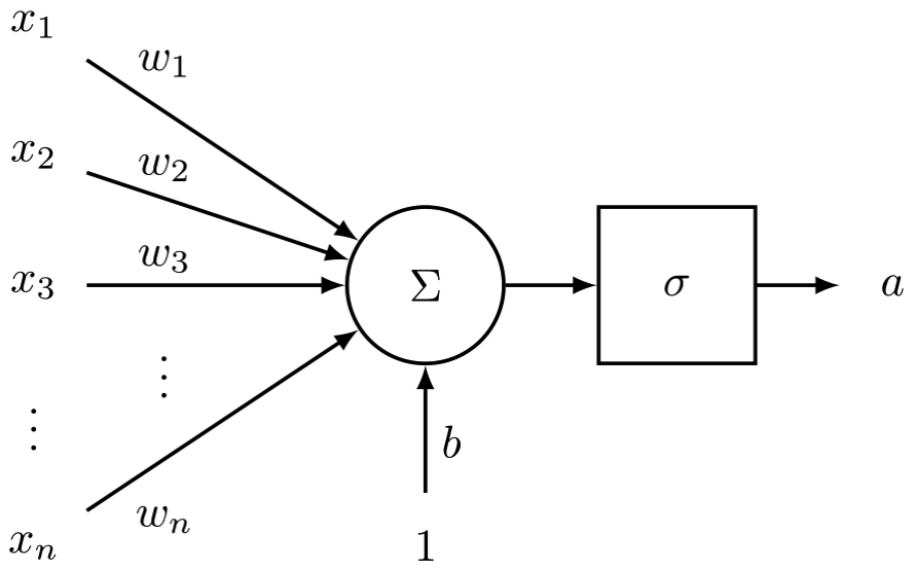
A neuron is a generic computational unit that takes n inputs and produces a single output. One of
the most popular choices for neurons is the "sigmoid" unit.

$$a = \frac{1}{1 + exp(-[w^T\ b] \cdot [x\ 1])}$$

where

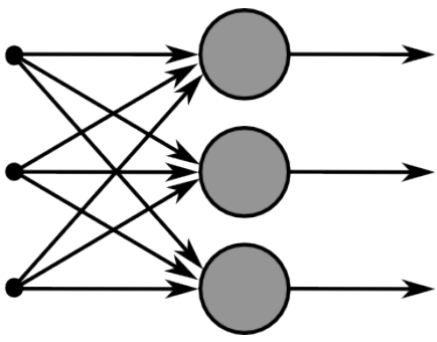
- a is the scalar activation output
- x is the input
- w is the weight matrix
- b is the bias term

The image below captures how in a sigmoid neuron, the input vector x is first scaled, summed, added to a bias unit, and then passed to the squashing sigmoid function.



2 A single layer of neurons

This image captures how multiple sigmoid units are stacked on the right, all of which receive the same input x.



The representation of the input x remains unchanged from the previous single neuron example. However, the weight matrix has become

$$W = \begin{bmatrix} - & w^{(1)T} & - \\ & \cdots & \\ - & w^{(m)T} & - \end{bmatrix} \in \mathbb{R}^{m \times n}$$

The bias term has become

$$b = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix} \in \mathbb{R}^m$$

The activation output can be written as

$$\sigma(z) = \begin{bmatrix} \frac{1}{1+\exp(z_1)} \\ \vdots \\ \frac{1}{1+\exp(z_m)} \end{bmatrix}$$

We can now write the output of scaling and biases as

$$z = Wx + b$$

The activations of the sigmoid function can then be written as

$$a = \begin{bmatrix} a^{(1)} \\ \vdots \\ a^{(m)} \end{bmatrix} = \sigma(z) = \sigma(Wx + b)$$

3 Intuition of hidden layer

Let us consider the following named entity recognition (NER) problem in NLP as an example:

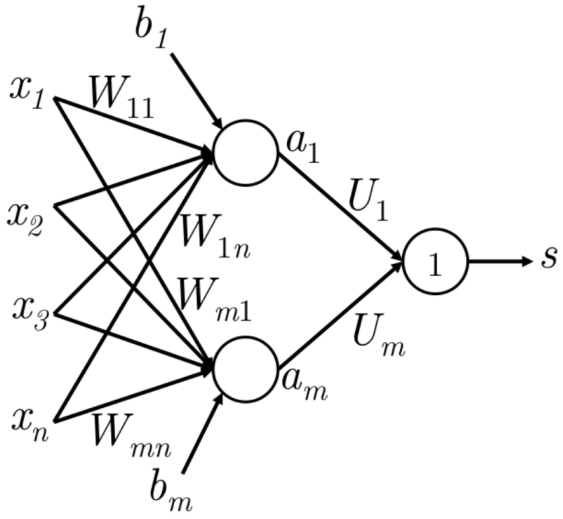
$$\text{Museums in Paris are amazing}$$

Here, we want to classify whether or not the center word "Paris" is a named-entity. In such cases, it is very likely that we would not just want to capture the presence of words in the window of word vectors but some other interactions between the words in order to make the classification.

For instance, maybe it should matter that "Museums" is the first word only if "in" is the second word. Such non-linear decisions can often not be captured by inputs fed directly to a Softmax function but instead require the scoring of the intermediate layer. We can thus use another matrix $U \in R^{m \times 1}$ to generate an unnormalized score for a classification task from the activations

$$s = U^T a = U^T f(Wx + b)$$

where f is the activation function.



Dimensions for a single hidden layer neural network:

If we represent each word using a 4-dimensional word vector and we use a 5-word window as input, then the input $x \in \mathbb{R}^{20}$. If we use 8 sigmoid units in the hidden layer and generate 1 score output from the activations, then

$$W \in \mathbb{R}^{8\times 20},\ b \in \mathbb{R}^{8},\ z \in \mathbb{R}^{8},\ U \in \mathbb{R}^{8}\ s \in \mathbb{R}$$

and

$$\begin{aligned} z &= Wx + b \\ a &= \sigma(z) \\ s &= U^T a \end{aligned}$$

4 Objective Function

In this example, we will discuss a popular error metric known as the `maximum margin objective`. The
idea behind using this objective is to ensure that the score computed for "true" labeled data points is higher than the score computed for "false" labeled data points.

Using the previous example, if we call the score computed for the "true" labeled window

$$\text{Museums in Paris are amazing}$$

as s and the score computed for the "false" labeled window

$$\text{Not all museums in Paris}$$

as s_c (subscripted as c to signify that the window is "corrupt").

Then, our objective function would be to maximize $(s - s_c)$ or to minimize $(s_c - s)$. However, we modify our objective to ensure that error is only computed if $s_c > s \Rightarrow (s_c - s) > 0$. The intuition behind doing this is that we only care the the "true" data point have a higher score than the "false" data point and that the rest does not matter. Thus, we want our error to be $(s_c - s)$ if $s_c > s$ else 0. Thus, our optimization objective is now:

$$\text{minimize } J = \max(s_c - s, 0)$$

However, the above optimization objective is risky in the sense that it does not attempt to create a margin of safety. We would want the "true" labeled data point to score higher than the "false" labeled data point by some positive margin Δ (more than just 0). In other words, we would want error to be calculated if $(s - s_c < \Delta)$ and not just when $(s - s_c < 0)$. Thus, we modify the

optimization objective:

$$\text{minimize } J = \max(\Delta + s_c - s, 0)$$

and the margin can be 1. Therefore, we can re-write it as:

$$\text{minimize } J = \max(1 + s_c - s, 0)$$

where

- $s_c = U^T f(Wx_c + b)$
- $s = U^T f(Wx + b)$

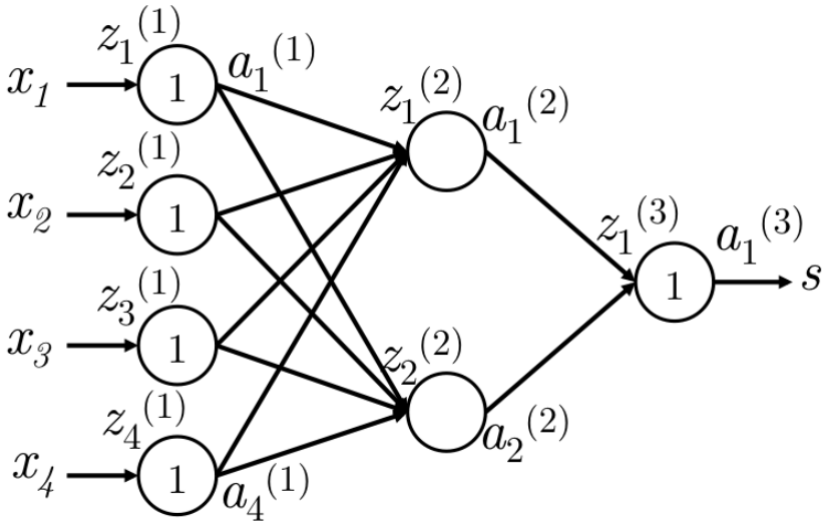
5 Training with Backpropagation – Elemental

In this section we discuss how we train the different parameters in the model when the cost $J = \max(1 + s_c - s, 0)$ is positive. No parameter updates are necessary if the cost is 0.

Since we typically update parameters using gradient descent (or a variant such as SGD), we typically need the gradient information for any parameter as required in the update equation:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla_{\theta^{(t)}} J$$

Backpropagation is technique that allows us to use the chain rule of differentiation to calculate loss gradients for any parameter used in the feed-forward computation on the model.

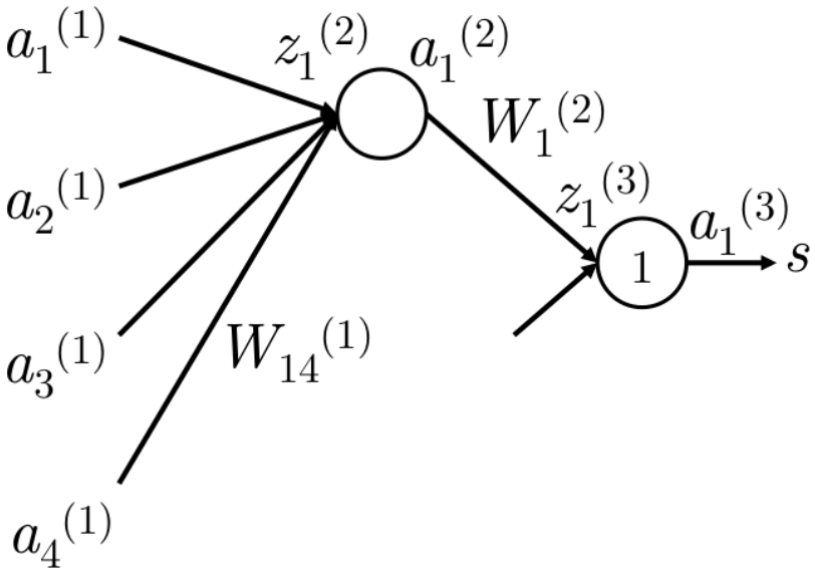


The figure above is a 4-2-1 neural network where neuron j on layer k receives input $z_j^{(k)}$ and produces activation output $\alpha_j^{(k)}$. Here, we use a neural network with a single hidden layer and a single unit output.

- x_i is an input to the neural network
- s is the output of the neural network
- Each layer (including the input and output layers) has neurons which receive an input and produce an output. The j^{-th} neuron of layer k receives the scalar input $z_j^{(k)}$ and produces the scalar activation output $a_j^{(k)}$
- $\delta_j^{(k)}$ denotes the backpropagated error calculated at $z_j^{(k)}$
- Layer 1 refers to the input layer and not the first hidden layer. For the input layer, $x_j = z_j^{(1)} = a^{(1)_j}$
- $W^{(k)}$ is the transfer matrix that maps the output from the k^{-th} layer to the input to the $(k_1)^{(-th)}$ layer. Thus, $W^{(1)} = W$ and $W^{(2)} = U$ to put this new generalized notation in

perspective of Section 1.3.

5.1 Backpropagation with chain rule



Suppose the cost $J = (1 + s_c - s)$ is positive and we want to perform the update of parameter $W_{14}^{(1)}$, we must realize that $W_{14}^{(1)}$ only contributes to $z_1^{(2)}$ and thus $a_1^{(2)}$. This fact is crucial to understanding backpropagation - backpropagated gradients are only affected by values they contribute to. $a_1^{(2)}$ is consequently used in the forward computation of score by multiplication with $W_1^{(2)}$.

We can see from the max-margin loss that:

$$\frac{\partial J}{\partial s} = -\frac{\partial J}{\partial s_c} = -1$$

Therefore we will work with $\frac{\partial s}{\partial W_{ij}^{(1)}}$ here for simplicity. Thus,

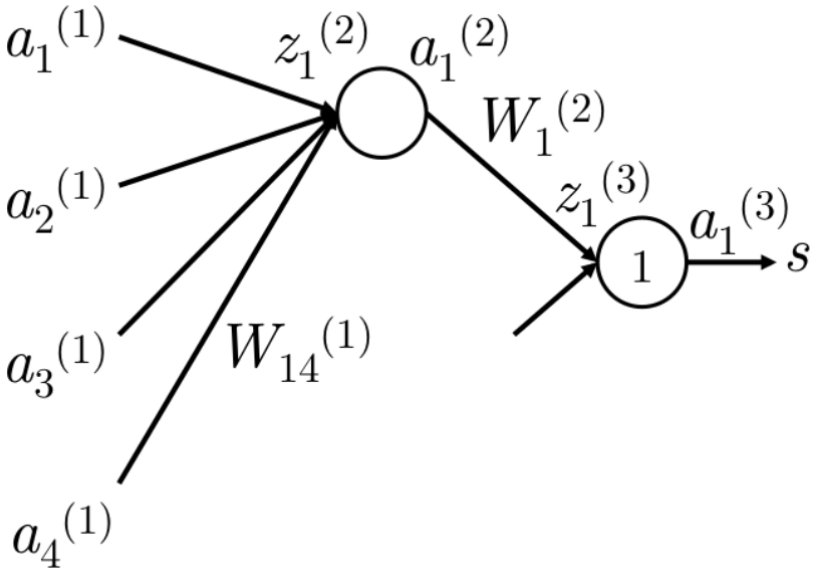
$$\frac{\partial s}{\partial W_{ij}^{(1)}} = \frac{\partial W^{(2)} a^{(2)}}{\partial W_{ij}^{(1)}} = \frac{\partial W_i^{(2)} a_i^{(2)}}{\partial W_{ij}^{(1)}} = W_i^{(2)} \frac{\partial a_i^{(2)}}{\partial W_{ij}^{(1)}}$$

$$\begin{aligned}
\Rightarrow W_i^{(2)} \frac{\partial a_i^{(2)}}{\partial W_{ij}^{(1)}} &= W_i^{(2)} \frac{\partial a_i^{(2)}}{\partial z_i^{(2)}} \frac{\partial z_i^{(2)}}{\partial W_{ij}^{(1)}} \\
&= W_i^{(2)} \frac{f\left(z_i^{(2)}\right)}{\partial z_i^{(2)}} \frac{\partial z_i^{(2)}}{\partial W_{ij}^{(1)}} \\
&= W_i^{(2)} f'\left(z_i^{(2)}\right) \frac{\partial z_i^{(2)}}{\partial W_{ij}^{(1)}} \\
&= W_i^{(2)} f'\left(z_i^{(2)}\right) \frac{\partial}{\partial W_{ij}^{(1)}} \left(b_i^{(1)} + a_1^{(1)} W_{i1}^{(1)} + a_2^{(1)} W_{i2}^{(1)} + a_3^{(1)} W_{i3}^{(1)} + a_4^{(1)} W_{i4}^{(1)}\right) \\
&= W_i^{(2)} f'\left(z_i^{(2)}\right) \frac{\partial}{\partial W_{ij}^{(1)}} \left(b_i^{(1)} + \sum_k a_k^{(1)} W_{ik}^{(1)}\right) \\
&= W_i^{(2)} f'\left(z_i^{(2)}\right) a_j^{(1)} \\
&= \delta_i^{(2)} \cdot a_j^{(1)}
\end{aligned}$$

We can see that the gradient reduces to the product $\delta_i^{(2)} \cdot a_j^{(1)}$

- $\delta_i^{(2)}$ is essentially the error propagating backwards from the i^{-th} neuron in layer 2
- The result of $a_j^{(1)}$ multiplied by W_{ij} is an input fed to i^{-th} neuron in layer 2. For example, the result of $a_4^{(1)}$ multiplied by $W_{14}^{(1)}$ is the input fed into the 1^{-th} neuron in layer 2.

5.2 Error sharing/distribution interpretation of backpropagation



Say we were to update $W_{14}^{(1)}$:

1. We start with an error signal of 1 propagating backwards from $a_1^{(3)}$.
2. We then multiply this error by the local gradient of the neuron which maps $z_1^{(3)}$ to $a_1^{(3)}$. This happens to be 1 in this case and thus, the error is still 1. This is now known as $\delta_1^{(3)} = 1$.
3. At this point, the error signal of 1 has reached $z_1^{(3)}$. We now need to distribute the error signal so that the "fair share" of the error reaches to $a_1^{(2)}$.
4. This amount is the $(\text{error signal at } z_1^{(3)} = \delta_1^{(3)}) \times W_1^{(2)} = 1 \times W_1^{(2)} = W_1^{(2)}$. Thus, the error at $a_1^{(2)} = W_1^{(2)}$. This is because $\frac{\partial z_1^{(3)}}{\partial a_1^{(2)}} = W_1^{(2)}$.
5. As we did in step 2, we need to move the error across the neuron which maps $z_1^{(2)}$ to $a_1^{(2)}$. We do this by multiplying the error signal at $a_1^{(2)}$ by the local gradient of the neuron which happens to be $f'(z_1^{(2)})$. This is because $\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} = f'(z_1^{(2)})$.
6. Thus, the error signal at $z_1^{(2)}$ is $f'(z_1^{(2)})W_1^{(2)}$. This is known as $\delta_1^{(2)}$.
7. Finally, we need to distribute the "fair share" of the error to $W_{14}^{(1)}$ by simply multiplying it by the input it was responsible for forwarding, which happens to be $a_4^{(1)}$. This is because $\frac{\partial z_1^{(2)}}{\partial W_{14}^{(1)}} = a_4^{(2)}$.
8. Thus, the gradient of the loss with respect to $W_{14}^{(1)}$ is calculated to be $a_4^{(1)} f'(z_1^{(2)}) W_1^{(2)}$.

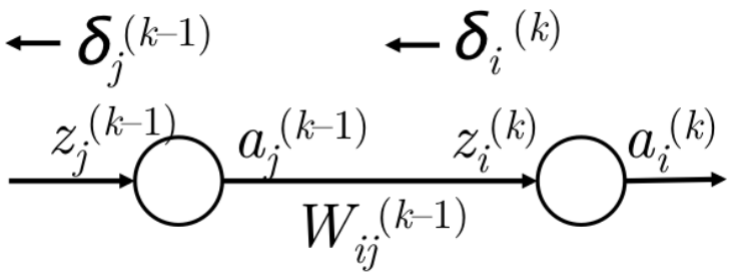
We can calculate error gradients with respect to a parameter in the network using either the chain rule of differentiation or using an error sharing and distributed flow approach – both of these approaches happen to do the exact same thing but it might be helpful to think about them one way or another.

5.3 Bias Updates

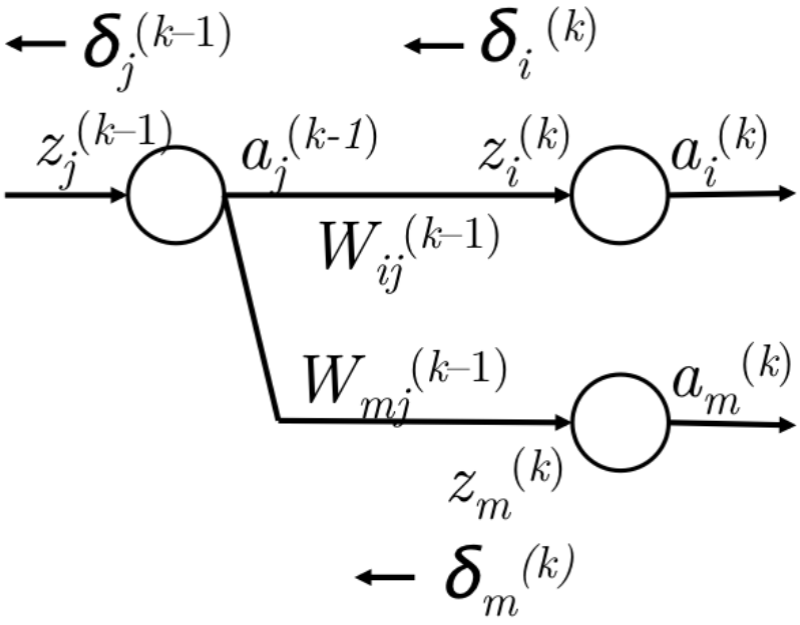
Bias terms (such as $b_1^{(1)}$) are mathematically equivalent to other weights contributing to the neuron input ($z_1^{(2)}$) as long as the input being forwarded is 1. As such, the bias gradients for neuron i on layer k is simply $\delta_i^{(k)}$. For instance, if we were updating $b_1^{(1)}$ instead of $W_{14}^{(1)}$ above, the gradient would simply be $f'(z_1^{(2)})W_1^{(2)}$. This is because $\frac{\partial z_1^{(2)}}{\partial b_1^{(1)}} = 1$.

5.4 Generalized steps to propagate $\delta^{(k)}$ to $\delta^{(k-1)}$

1. We have error $\delta^{(k)}$ propagating backwards from $z_i^{(k)}$, i.e. neuron i at layer k.



2. We propagate this error backwards to $a_j^{(k-1)}$ by multiplying $\delta^{(k)}$ by the path weight $W_{ij}^{(k-1)}$.
3. Thus, the error received at $a_j^{(k-1)}$ is $\delta_i^{(k)} W_{ij}^{(k-1)}$.
4. However, $a_j^{(k-1)}$ may have been forwarded to multiple nodes in the next layer. It should receive responsibility for errors propagating backward from node m in layer k too, using the exact same mechanism.



5. Thus, error received at $a_j^{(k-1)}$ is $\delta_i^{(k)} W_{ij}^{(k-1)} + \delta_m^{(k)} W_{mj}^{(k-1)}$.
6. In fact, we can generalize this to be $\sum_i \delta_i^{(k)} W_{ij}^{(k-1)}$.
7. Now that we have the correct error at $a_j^{(k-1)}$, we move it across neuron j at layer $k-1$ by multiplying with the local gradient $f'(z_j^{(k-1)})$.
8. Thus, the error that reaches $z_j^{(k-1)}$, called $\delta_j^{(k-1)}$ is $f'(z_j^{(k-1)}) \sum_i \delta_i^{(k)} W_{ij}^{(k-1)}$.

6 Training with Backpropagation - Vectorized

So far, we discussed how to calculate gradients for a given parameter in the model. Here we will generalize the approach above so that we update weight matrices and bias vectors all at once. Note that these are simply extensions of the above model that will help build intuition for the way error propagation can be done at a matrix vector level.

For a given parameter $W_{ij}^{(k)}$, we identified that the error gradient is simply $\delta_i^{(k+1)} \cdot a_j^{(k)}$. As a reminder, $W^{(k)}$ is the matrix that maps $a^{(k)}$ to $z^{(k+1)}$. We can thus establish that the error gradient for the entire matrix $W^{(k)}$ is:

$$\nabla_{W^{(k)}} = \begin{bmatrix} \delta_1^{(k+1)} a_1^{(k)} & \delta_1^{(k+1)} a_2^{(k)} & \cdots \\ \delta_2^{(k+1)} a_1^{(k)} & \delta_2^{(k+1)} a_2^{(k)} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} = \delta^{(k+1)} a^{(k)T}$$

Thus, we can write an entire matrix gradient using the outer product of the error vector propagating into the matrix and the activations forwarded by the matrix.

Now, we will see how we can calculate the error vector $\delta^{(k)}$. We establish earlier that $\delta_j^{(k)} = f'(z_j^{(k)}) \sum_i \delta_i^{(k+1)} W_{ij}^{(k)}$. This can easily generalize to matrices such that:

$$\delta^{(k)} = f'\left(z^{(k)}\right) \circ \left(W^{(k)T} \delta^{(k+1)}\right)$$

In the above formulation, the $\circ$ operator corresponds to an element wise product between elements of vectors ($\circ : \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}^N$).

6.1 Computational efficiency

Having explored element-wise updates as well as vector-wise updates, we must realize that the vectorized implementations run substantially faster in scientific computing environments such as MATLAB or Python (using NumPy/SciPy packages). Thus, we should use vectorized implementation in practice.

Furthermore, we should also reduce redundant calculations in backpropagation - for instance, notice that $\delta^{(k)}$ depends directly on $\delta^{(k+1)}$. Thus, we should ensure that when we update $W^{(k)}$ using $\delta^{(k+1)}$, we save $\delta^{(k+1)}$ to later derive $\delta^{(k)}$ – and we then repeat this for $(k-1)\ldots(1)$. Such a recursive procedure is what makes backpropagation a computationally affordable procedure.

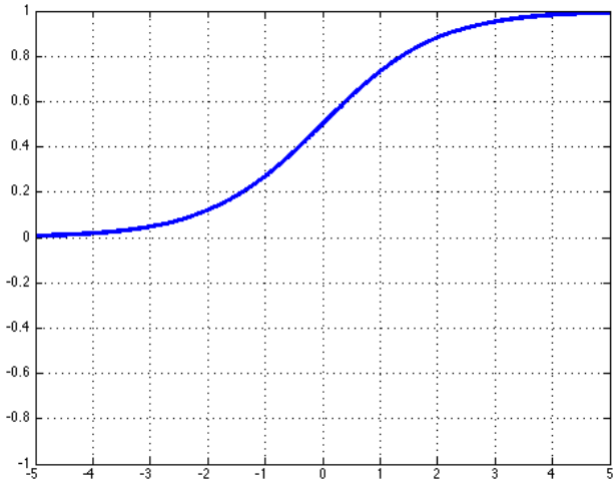
7 Neural network tricks

7.1 Activation functions

7.1.1 Sigmoid

$$\sigma(z) = \frac{1}{1 + exp(-z)}$$

where $\sigma(z) \in (0, 1)$.



The gradient of $\sigma(z)$ is

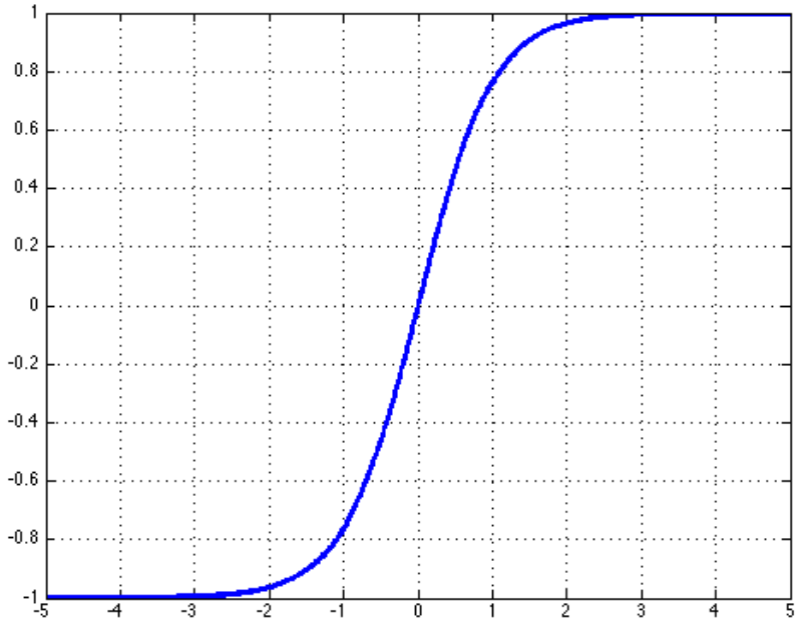
$$\sigma'(z) = \frac{-exp(-z)}{1 + exp(-z)} = \sigma(z)(1 - \sigma(z))$$

7.1.2 Tanh

The tanh function is an alternative to the sigmoid function that is often found to converge faster in practice. The primary difference between tanh and sigmoid is that tanh output ranges from −1 to 1 while the sigmoid ranges from 0 to 1.

$$tanh(z) = \frac{exp(z) - exp(-z)}{exp(z) + exp(-z)} = 2\sigma(2z) - 1$$

where $tanh(z) \in (-1, 1)$.



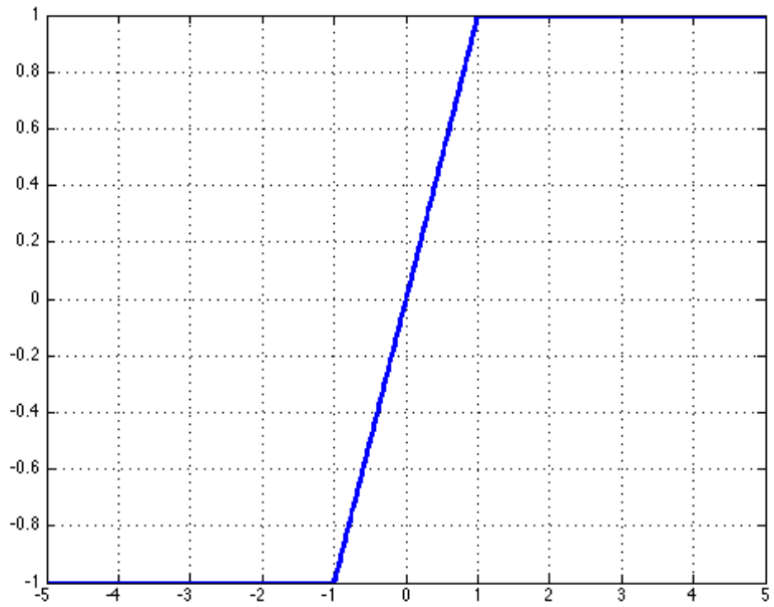
The gradient of tanh(z) is

$$tanh'(z) = 1 - (\frac{exp(z) - exp(-z)}{exp(z) + exp(-z)})^2 = 1 - tanh^2(z)$$

7.1.3 Hard tanh

The hard tanh function is sometimes preferred over the tanh function since it is **computationally cheaper**. It does however saturate for magnitudes of z greater than 1. The activation of the hard tanh is:

$$\text{hardtanh}(z) = \begin{cases} -1 : z < -1 \\ z : -1 \leq z \leq 1 \\ 1 : z > 1 \end{cases}$$



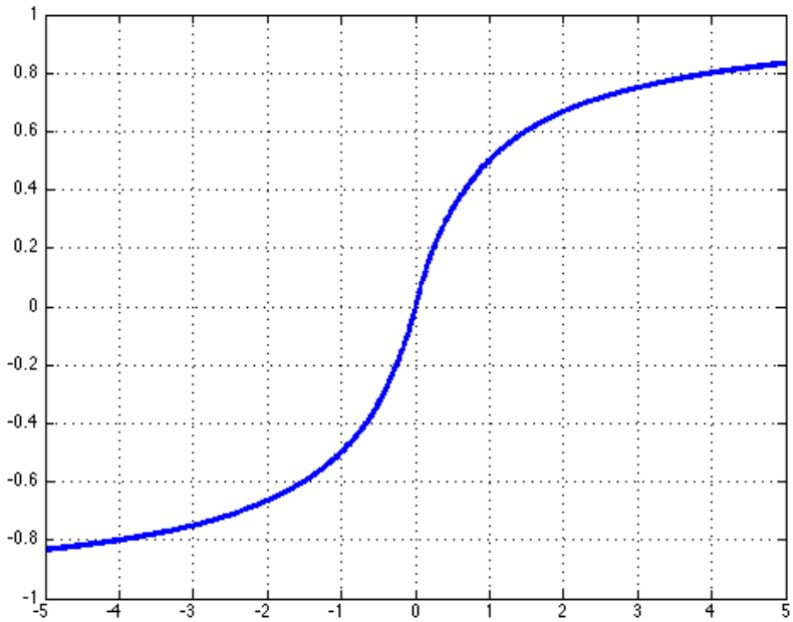
The derivative can also be expressed in a piecewise functional form:

$$\text{hardtanh}'(z) = \begin{cases} 1 : -1 \leq z \leq -1 \\ 0 : \text{otherwise} \end{cases}$$

7.1.4 Soft sign

The soft sign function is another nonlinearity which can be considered an alternative to tanh since it too does not saturate as easily as hard clipped functions:

$$\text{softsign}(z) = \frac{z}{1 + |z|}$$



The derivative is expressed as:

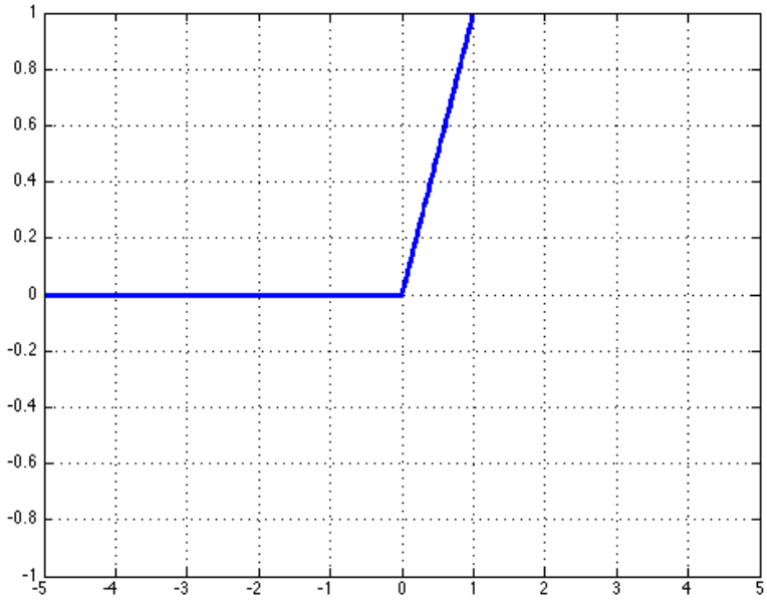
$$\text{softsign}'(z) = \frac{sgn(z)}{(1+z)^2}$$

where sgn is the signum function which returns ±1 depending on the sign of z.

7.1.5 ReLU

The ReLU (Rectified Linear Unit) function is a popular choice of activation since it does not saturate even for larger values of z and has found much success in computer vision applications:

$$\text{rect}(z) = max(z, 0)$$



The derivative is then the piecewise function:

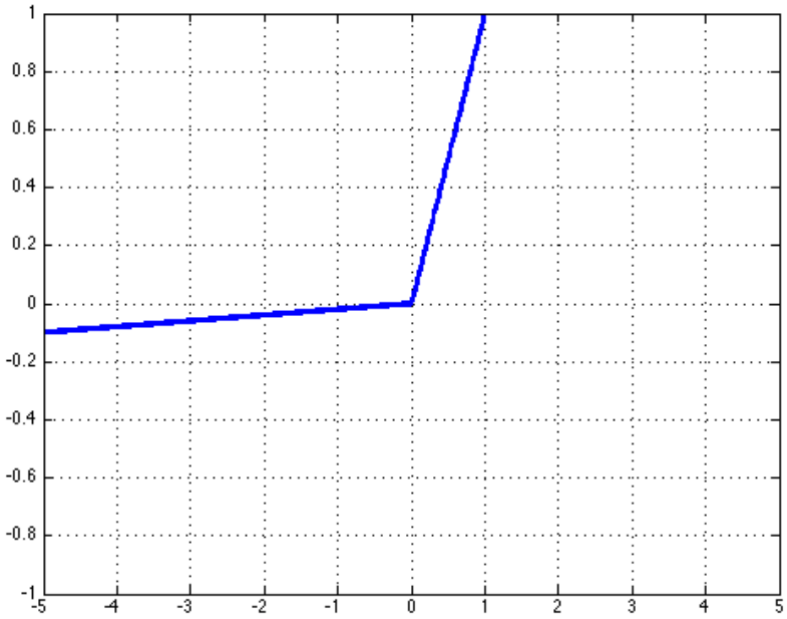
$$\text{rect}'(z) = \begin{cases} 1 : z > 0 \\ 0 : \text{otherwise} \end{cases}$$

7.1.6 Leaky ReLU

Traditional ReLU units by design do not propagate any error for non-positive z – the leaky ReLU modifies this such that a small error is allowed to propagate backwards even when z is negative:

$$\text{leaky}(z) = max(z, k \cdot z)$$

where $0 < k < 1$.



This way, the derivative is representable as:

$$\text{leaky}'(z) = \begin{cases} 1 : z > 0 \\ k : \text{otherwise} \end{cases}$$

7.2 Data preprocessing

7.2.1 Mean Subtraction

Given a set of input data X, it is customary to zero-center the data by subtracting the mean feature vector of X from X.

An important point is that in practice, the mean is calculated only across the training set, and this mean is subtracted from the training, validation, and testing sets. This way we can prevent data leakage.

7.2.2 Normalization

Another frequently used technique (though perhaps less so than mean subtraction) is to scale every input feature dimension to have similar ranges of magnitudes. This is useful since input features are often measured in different "units", but we often want to initially consider all features as equally important.

The way we accomplish this is by simply dividing the features by their respective standard deviation calculated across the training set.

7.2.3 Whitening

Not as commonly used as mean-subtraction + normalization, whitening essentially converts the data to a have an identity covariance matrix – that is, features become uncorrelated and have a variance
of 1.

This is done by first mean-subtracting the data, as usual, to get X'. We can then take the Singular Value Decomposition (SVD) of X' to get matrices U, S, V. We then compute UX' to project X' into the
basis defined by the columns of U. We finally divide each dimension of the result by the corresponding singular value in S to scale our data appropriately (if a singular value is zero, we

can just divide by a
small number instead).

7.3 Parameter Initialization (Xavier)

A good starting strategy is to initialize the weights to small random numbers normally distributed around 0 – and in practice, this often words acceptably well.

However, in `Understanding the difficulty of training deep feedforward neural networks (2010)`, Xavier et al study the effect of different weight and bias initialization schemes on training dynamics. The empirical findings suggest that for **sigmoid and tanh** activation units, faster convergence and lower error rates are achieved when the weights of a matrix $W \in \mathbb{R}^{n^{(l+1)} \times n^{(l)}}$ are initialized randomly with a uniform distribution as follows:

$$W \sim U\left[-\sqrt{\frac{6}{n^{(l)} + n^{(l+1)}}}, \sqrt{\frac{6}{n^{(l)} + n^{(l+1)}}} \right]$$

where

- $n^{(l)}$ is the number of input units to W (fan-in)
- $n^{(l+1)}$ is the number of output units from W (fan-out)

In this parameter initialization scheme, bias units are initialized to 0.

This approach attempts to maintain activation variances as well as backpropagated gradient variances across layers. Without such initialization, the gradient variances (which are a proxy for information) generally decrease with backpropagation across layers.

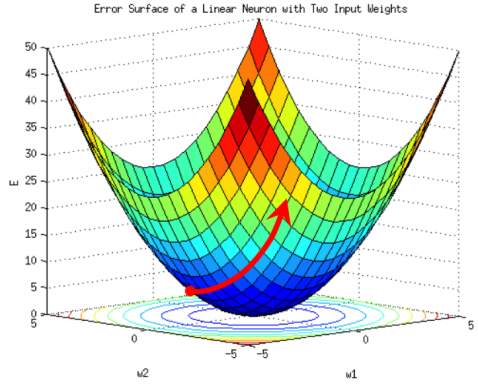
7.4 Learning Strategies

7.4.1 Hand-set learning rates

The rate/magnitude of model parameter updates during training can be controlled using the learning rate. In the following naïve Gradient Descent formulation, α is the learning rate:

$$\theta^{new} = \theta^{old} - \alpha \nabla_\theta J_t(\theta)$$

When learning rate α is too large, we might experience that the loss function actually diverges because the
parameters update causes the model to overshoot the convex minima. In non-convex models (most of those we work with), the outcome of a large learning rate is unpredictable, but the chances of diverging loss functions are very high.



When learning rate α is too small, we might not converge in a reasonable amount of time, or might get caught in local minima.

Since training is the most expensive phase in a deep learning system, some research has attempted to improve this naïve approach to setting learning learning rates. For instance, Ronan Collobert scales the learning rate of a weight W_{ij} (where $W \in \mathbb{R}^{n^{(l+1)} \times n^{(l)}}$) by the inverse square root of the fan-in of the neuron ($n^{(l)}$).

There are several other techniques that have proven to be effective as well – one such method is **annealing**, where, after several iterations, the learning rate is reduced in some way – this method ensures that we start off with a high learning rate and approach a minimum quickly; as we get closer to the minimum, we start lowering our learning rate so that we can find the optimum under a more fine-grained scope.

- A common way to perform annealing is to reduce the learning rate α by a factor x after every n iterations of learning.
- Exponential decay is also common, where, the learning rate α at iteration t is given by $\alpha(t) = \alpha_0 e^{-kt}$, where α_0 is the initial learning rate, and k is a hyperparameter.
- Another approach is to allow the learning rate to decrease over time such that $\alpha(t) = \frac{\alpha_0 \tau}{\max(t, \tau)}$ where α_0 is a tunable parameter and represents the starting learning rate. τ is also a tunable parameter and represents the time at which the learning rate should start reducing. In practice, this method has been found to work quite well.

7.4.2 Momentum updates

Momentum methods enables adaptive gradient descent without the need of hand-set learning rates. It is a variant of gradient descent inspired by the study of dynamics and motion in physics, attempting to use the "velocity" of updates as a more effective update scheme. Pseudocode for momentum updates is shown below:

```
# Computes a standard momentum update
# on parameters x
v = mu*v - alpha*grad_x
x += v
```

7.5 Adaptive Optimization Methods

7.5.1 AdaGrad

AdaGrad is an implementation of standard stochastic gradient descent (SGD) with one key difference: the learning rate can vary for each parameter.

The learning rate for each parameter depends on the history of gradient updates of that parameter in a way such that parameters with a scarce history of updates are updated faster using a larger learning rate. In other words, parameters that have not been updated much in the past are likelier to have higher learning rates now. Formally:

$$\theta_{t,i} = \theta_{t-1,i} - \frac{\alpha}{\sqrt{\sum_{\tau=1}^{t} g_{\tau,i}^2}} g_{t,i} \text{ where } g_{t,i} = \frac{\partial}{\partial \theta_i^t} J_t(\theta)$$

In this technique, we see that if the RMS of the history of gradients is extremely low, the learning rate is very high. A simple implementation of this technique is:

```
# Assume the gradient dx and parameter vector x
cache += dx**2
x += - learning_rate * dx / np.sqrt(cache + 1e-8)
```

7.5.2 RMSProp and Adam

`RMSProp`:

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

`Adam`:

```
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
x += - learning_rate * m / (np.sqrt(v) + eps)
```

RMSProp is a variant of AdaGrad that utilizes a moving average of squared gradients – in particular, unlike AdaGrad, its updates do not become monotonically smaller. The Adam update rule is in turn a variant of RMSProp, but with the addition of momentum like updates

.