CS 188 Introduction to Artificial Intelligence
Fall 2017 Note 10

# Perceptron

## Linear Classifiers

In the previous note, we explored using a Naive Bayes classifier. The core idea behind Naive Bayes is to extract certain attributes of the training data called features and then estimate the probability of a label given the features: $P(y|f_1, f_2, ...f_n)$. Thus, given a new data point, we could then extract the corresponding features, and classify the new data point with the label with the highest probability given the features. This all, however, this requires us to estimate distributions, which we did with MLE. What if instead we decided not to estimate the probability distribution? Lets start by looking at a simple linear classifier, which we can use for **binary classification**, which is when the label has two possibilities, positive or negative.

The basic idea of a **linear classifier** is to do classification using a linear combination of the features– a value which we call the **activation**. Concretely, the activation function takes in a data point, multiplies each feature of our data point, $f_i(x)$, by a corresponding weight, $w_i$, and outputs the sum of all the resulting values. In vector form, we can also write this as a dot product of our weights as a vector, $\mathbf{w}$, and our featurized data point as a vector $\mathbf{f}(x)$:

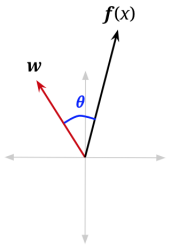
$$\text{activation}_w(x) = \sum_i w_i f_i(x) = \mathbf{w}^T \mathbf{f}(x) = \mathbf{w} \cdot \mathbf{f}(x)$$

How does one do classification using the activation? For binary classification, when the activation of a data point is positive, we classify that data point with the positive label, and if it is negative, we classify with the negative label.

$$\text{classify}(x) = \begin{cases} + & \text{if activation}_w(x) > 0 \\ - & \text{if activation}_w(x) < 0 \end{cases}$$

To understand this geometrically, let us reexamine the vectorized activation function. Using the Law of Cosines, we can rewrite the dot product as follows, where $\|\cdot\|$ is the magnitude operator and $\theta$ is the angle between $\mathbf{w}$ and $\mathbf{f}(x)$:

$$\text{activation}_w(x) = \mathbf{w} \cdot \mathbf{f}(x) = \|\mathbf{w}\|\|\mathbf{f}(\mathbf{x})\| \cos(\theta)$$




Since magnitudes are always non-negative, and our classification rule looks at the sign of the activation, the

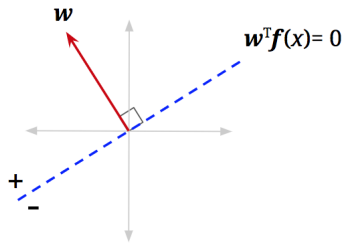
only term that matters for determining the class is $\cos(\theta)$.

$$\text{classify}(x) = \begin{cases} + & \text{if } cos(\theta) > 0 \\ - & \text{if } cos(\theta) < 0 \end{cases}$$

We, therefore, are interested in when $cos(\theta)$ is negative or postive. It is easily seen that for $\theta < \frac{\pi}{2}$, $cos(\theta)$ will be somewhere in the interval $(0,1]$, which is positive. For $\theta > \frac{\pi}{2}$, $cos(\theta)$ will be somewhere in the interval $[-1,0)$, which is negative. You can confirm this by looking at a unit circle. Essentially, our simple linear classifier is checking to see if the feature vector of a new data point roughly "points" in the same direction as a predefined weight vector and applies a positive label if it does.

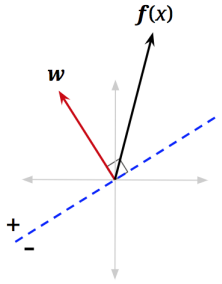
$$\text{classify}(x) = \begin{cases} + & \text{if } \theta < \frac{\pi}{2} \quad \text{(i.e. when } \theta \text{ is less than } 90°\text{, or acute)} \\ - & \text{if } \theta > \frac{\pi}{2} \quad \text{(i.e. when } \theta \text{ is greater than } 90°\text{, or obtuse)} \end{cases}$$

Up to this point, we haven't considered the points where $\text{activation}_w(x) = \mathbf{w}^T\mathbf{f}(x) = 0$. Following all the same logic, we will see that $cos(\theta) = 0$ for those points. Furthermore, $\theta = \frac{\pi}{2}$ (i.e $\theta$ is 90°) for those points. In otherwords, these are the data points with feature vectors that are orthogonal to $\mathbf{w}$. We can add a dotted blue line, orthogonal to $\mathbf{w}$, where any feature vector that lies on this line will have activation equaling 0.

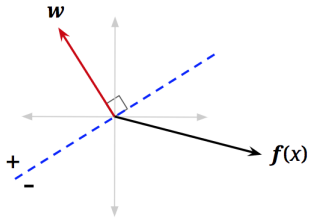



**Decision Boundary**

We call this blue line the **decision boundary** because it is the boundary that separates the region where we classify data points as positive from the region of negatives. In higher dimensions, a linear decision boundary is generically called a **hyperplane**. A hyperplane is a linear surface that is one dimesion lower than the latent space, thus dividing the surface in two. For general classifiers (non-linear ones), the decision boundary may not be linear, but is simply defined as surface in the space of feature vectors that separates the classes. To classify points that end up on the decision boundary, we can apply either label since both classes are equally valid (in the algorithms below, we'll classify points on the line as positive).




*x* **classified into positive class**




*x* **classified into negative class**

# Binary Perceptron

Great, now you know how linear classifiers work, but how do we build a good one? When building a classifier, you start with data, which are labeled with the correct class, we call this the **training set**. You build a classifier by evaluating it on the training data, comparing that to you training labels, and adjusting the parameters of your classifier until you reach your goal.

Let's explore one specific implementation of a simple linear classifier: the binary perceptron. The perceptron is a binary classifier–though it can be extended to work on more than two classes. The goal of the binary perceptron is to find a decision boundary that perfectly separates the training data. In other words, we're seeking the best possible weights– the best **w**– such that any featured training point that is multiplied by the weights, can be perfectly classified.

**The Algorithm**
The perceptron algorithm works as follows:

1. Initialize all weights to 0: $\mathbf{w} = \mathbf{0}$
2. For each training sample, with features $\mathbf{f}(x)$ and true class label $y^* \in \{-1, +1\}$, do:
   (a) Classify the sample using the current weights, let $y$ be the class predicted by your current **w**:
   $$y = \text{classify}(x) = \begin{cases} +1 & \text{if activation}_w(x) = \mathbf{w}^T\mathbf{f}(x) > 0 \\ -1 & \text{if activation}_w(x) = \mathbf{w}^T\mathbf{f}(x) < 0 \end{cases}$$
   (b) Compare the predicted label $y$ to the true label $y^*$:
   - If $y = y^*$, do nothing
   - Otherwise, if $y \neq y^*$, then update your weights: $\mathbf{w} \leftarrow \mathbf{w} + y^*\mathbf{f}(x)$
3. If you went through **every** training sample without having to update your weights (all samples predicted correctly), then terminate. Else, repeat step 2

**Updating weights**
Let's examine and justify the procedure for updating our weights. Recall that in step 2b above, when our classifier is right, nothing changes. But when our classifier is wrong, the weight vector is updated as follows:

$$\mathbf{w} \leftarrow \mathbf{w} + y^*\mathbf{f}(x)$$

where $y^*$ is the true label, which is either 1 or -1, and $x$ is the training sample which we mis-classified. You can interpret this update rule to be:

| | |
|---|---|
| **Case 1** : mis-classified positive as negative | $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{f}(x)$ |
| **Case 2** : mis-classified negative as positve | $\mathbf{w} \leftarrow \mathbf{w} - \mathbf{f}(x)$ |

Why does this work? One way to look at this is to see it as a balancing act. Mis-classification happens either when the activation for a training sample is much smaller than it should be (causes a Case 1 misclassification) or much larger than it should be (causes a Case 2 misclassification).

Consider Case 1, where activation is negative when it should be positive. In otherwords, the activation is too small. How we adjust **w** should strive to fix that and make the activation larger for that training sample. To

convince yourself that our update rule $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{f}(x)$ does that, let us update $\mathbf{w}$ and see how the activation changes.

$$\text{activation}_{w+x}(x) = (\mathbf{w} + \mathbf{f}(x))^T \mathbf{x} = \mathbf{w}^T \mathbf{f}(x) + \mathbf{f}(x)^T \mathbf{f}(x) = \text{activation}_w(x) + \mathbf{f}(x)^T \mathbf{f}(x)$$

Using our update rule, we see that the new activation increases by $\mathbf{f}(x)^T \mathbf{f}(x)$, which is a postive number, therefore showing that our update makes sense. Activation is getter larger– closer to becoming positive. You can repeat the same logic for when the classifier is mis-classifying because the activation is too large (activation is positive when it should be negative). You'll see that the update will cause the new activation to decrease by $\mathbf{f}(x)^T \mathbf{f}(x)$, thus getting smaller and closer to classifying correctly.

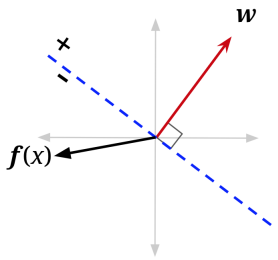
While this makes it clear why we are adding and subtracting *something*, why would we want to add and subtract our sample point's features? A good way to think about it, is that the weights aren't the only thing that determines this score. The score is determined by multiplying the weights by the relevant sample. This means that certain parts of a sample contribute more than others. Consider the following situation where $x$ is a training sample we are given with true label $y^* = -1$:

$$\mathbf{w}^T = \begin{bmatrix} 2 & 2 & 2 \end{bmatrix}, \mathbf{f}(x) = \begin{bmatrix} 4 \\ 0 \\ 1 \end{bmatrix} \qquad \text{activation}_w(x) = (2*4) + (2*0) + (2*1) = 10$$

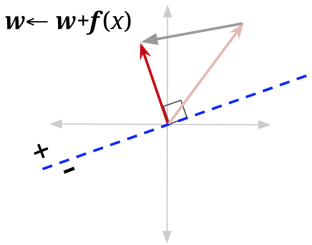
We know that our weights need to be smaller because activation needs to be negative to classify correctly. We don't want to change them all the same amount though. You'll notice that the first element of our sample, the 4, contributed much more to our score of 10 than the third element, and that the second element did not contribute at all. An appropriate weight update, then, would change the first weight alot, the third weight a little, and the second weight should not be changed at all. After all, the second and third weights might not even be that broken, and we don't fix want to fix what isn't broken!

When thinking about a good way to change our weight vector in order to fulfill the above desires, it turns out just using the sample itself does in fact do what we want; it changes the first weight by alot, the third weight by a little, and doesn't change the second weight at all! *Note that this is similar to our updates in approximate Q-learning, which also updated weights by multiples of the feature vectors.*

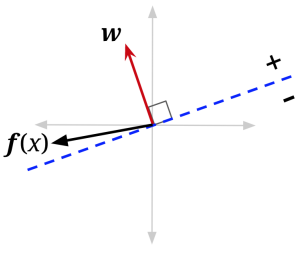
A visualization may also help. In the figure below, $\mathbf{f}(x)$ is the feature vector for a data point with positive class ($y^* = +1$) that is currently misclassified – it lies on the wrong side of the decision boundary defined by "old $\mathbf{w}$". Adding it to the weight vector produces a new weight vector which has a smaller angle to $\mathbf{f}(x)$. It also shifts the decision boundary. In this example, it has shifted the decision boundary enough so that $x$ will now be correctly classified (note that the mistake won't always be fixed – it depends on the size of the weight vector, and how far over the boundary $\mathbf{f}(x)$ currently is).




**Mis-classifying x with old w**




**Updating w**



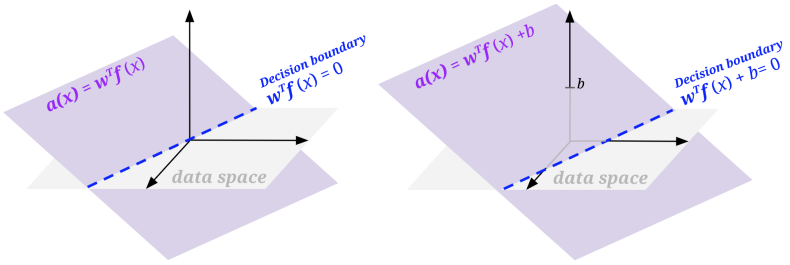

**Updated classification of x**

**Bias**

If you tried to implement a perceptron based on what has been mentioned thus far, you will notice one particularly unfriendly quirk. Any decision boundary that you end up drawing will be crossing the origin. Basically, your perceptron can only produce a decision boundary that could be represented by the function $\mathbf{w}^\top \mathbf{f}(x) = 0$, $\mathbf{w}, \mathbf{f}(x) \in \mathbb{R}^n$. The problem is, even among problems where there is a linear decision boundary that separates the positive and negative classes in the data, that boundary may not go through the origin, and we want to be able to draw those lines.

To do so, we will modify our feature and weights to add a bias term: add a feature to your sample feature vectors that is always 1, and add an extra weight for this feature to your weight vector. Doing so essentially allows us to produce a decision boundary representable by $\mathbf{w}^\top \mathbf{f}(x) + b = 0$, where $b$ is the weighted bias term (i.e. 1 * the last weight in the weight vector).

Geometrically, we can visualize this by thinking about what the activation function looks like when it is $\mathbf{w}^\top \mathbf{f}(x)$ and when there is a bias $\mathbf{w}^\top \mathbf{f}(x) + b$. To do so, we need to be one dimension higher than the space of our featurized data (labeled data space in the figures below). In all the above sections, we had only been looking at a flat view of the data space.




**Without bias** **With bias**

**Example**

Up until now, we have been focused on alot of reading, let's shake things up a little and see an example, and run the perceptron algorithm step by step.

Let's run one pass through the data with the perceptron algorithm, taking each data point in order. We'll start with the weight vector $[w_0, w_1, w_2] = [-1, 0, 0]$ (where $w_0$ is the weight for our bias feature, which remember is always 1).

**Training Set**

| # | $f_1$ | $f_2$ | $y^*$ |
|---|---|---|---|
| 1 | 1 | 1 | - |
| 2 | 3 | 2 | + |
| 3 | 2 | 4 | + |
| 4 | 3 | 4 | + |
| 5 | 2 | 3 | - |

**Single Perceptron Update Pass**

| step | Weights | Score | Correct? | Update |
|---|---|---|---|---|
| 1 | [-1, 0, 0] | $-1 \cdot 1 + 0 \cdot 1 + 0 \cdot 1 = -1$ | yes | none |
| 2 | [-1, 0, 0] | $-1 \cdot 1 + 0 \cdot 3 + 0 \cdot 2 = -1$ | no | +[1, 3, 2] |
| 3 | [0, 3, 2] | $0 \cdot 1 + 3 \cdot 2 + 2 \cdot 4 = 14$ | yes | none |
| 4 | [0, 3, 2] | $0 \cdot 1 + 3 \cdot 3 + 2 \cdot 4 = 17$ | yes | none |
| 5 | [0, 3, 2] | $0 \cdot 1 + 3 \cdot 2 + 2 \cdot 3 = 12$ | no | -[1, 2, 3] |
| 6 | [-1, 1, -1] | | | |

We'll stop here, but in actuality this algorithm would run for many more passes through the data before all the data points are classified correctly in a single pass.

# Multiclass Perceptron

The perceptron presented above is a binary classifier, but we can extend it to account for multiple classes rather easily. The main difference is in how we set up weights and how we update said weights. For the binary case, we had one weight vector, which had a dimension equal to the number of features (plus the bias feature). For the multi-class case, we will have one weight vector for each class, so in the 3 class case, we have 3 weight vectors. In order to classify a sample, we compute a score for each class by taking the dot product of the feature vector with each of the weight vectors. Whichever class yields the highest score is the one we choose as our prediction.

For example, consider the 3-class case. Let our sample have features $\mathbf{f}(x) = \begin{bmatrix} -2 & 3 & 1 \end{bmatrix}$ and our weights for classes 0, 1, and 2 be:

$$\mathbf{w}_0 = \begin{bmatrix} -2 & 2 & 1 \end{bmatrix}$$
$$\mathbf{w}_1 = \begin{bmatrix} 0 & 3 & 4 \end{bmatrix}$$
$$\mathbf{w}_2 = \begin{bmatrix} 1 & 4 & -2 \end{bmatrix}$$

Taking dot products for each class gives us scores $s_0 = 11, s_1 = 13, s_2 = 8$. We would thus predict that $x$ belongs to class 1.

An important thing to note is that in actual implementation, we do not keep track of the weights as separate structures, we usually stack them on top of each other to create a weight matrix. This way, instead of doing as many dot products as there are classes, we can instead do a single matrix-vector multiplication. This tends to be much more efficient in practice (because matrix-vector multiplication usually has a highly optimized implementation).

In our above case, that would be:

$$W = \begin{bmatrix} -2 & 2 & 1 \\ 0 & 3 & 4 \\ 1 & 4 & -2 \end{bmatrix}, x = \begin{bmatrix} -2 \\ 3 \\ 1 \end{bmatrix}$$

And our label would be:

$$\arg\max(Wx) = \arg\max\left( \begin{bmatrix} 11 \\ 13 \\ 8 \end{bmatrix} \right) = 1$$

Along with the structure of our weights, our weight update also changes when we move to a multi-class case. If we correctly classify our data point, then do nothing just like in the binary case. If we chose incorrectly, say we chose class $y \neq y^*$, then we add the feature vector to the weight vector for the true class to $y^*$, and subtract the feature vector from the weight vector corresponding to the predicted class $y$. In our above example, let's say that the correct class was class 2, but we predicted class 1. We would now take the weight vector corresponding to class 1 and subtract $x$ from it,

$$w_1 = \begin{bmatrix} 0 & 3 & 4 \end{bmatrix} - \begin{bmatrix} -2 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 3 \end{bmatrix}$$

Next we take the weight vector corresponding to the correct class, class 2 in our case, and add $x$ to it:

$$w_2 = \begin{bmatrix} 1 & 4 & -2 \end{bmatrix} + \begin{bmatrix} -2 & 3 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 7 & -1 \end{bmatrix}$$

What this amounts to is 'rewarding' the correct weight vector, 'punishing' the misleading, incorrect weight vector, and leaving alone an other weight vectors. With the difference in the weights and weight updates taken into account, the rest of the algorithm is essentially the same; cycle through every sample point, updating weights when a mistake is make, until you stop making mistakes.

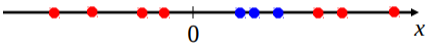
In order to incorporate a bias term, do the same as we did for binary perceptron – add an extra feature of 1 to every feature vector, and an extra weight for this feature to every class's weight vector (this amounts to adding an extra column to the matrix form).

# Toward Neural Networks

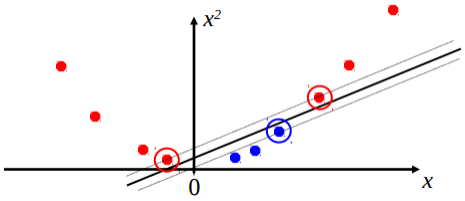
## Non-linear Separators

We know how to construct a model that learns a linear boundary for binary classification tasks. This is a powerful technique, and one that works well when the underlying optimal decision boundary is itself linear. However, many practical problems involve the need for decision boundaries that are nonlinear in nature, and our linear perceptron model isn't expressive enough to capture this relationship.

Consider the following set of data:



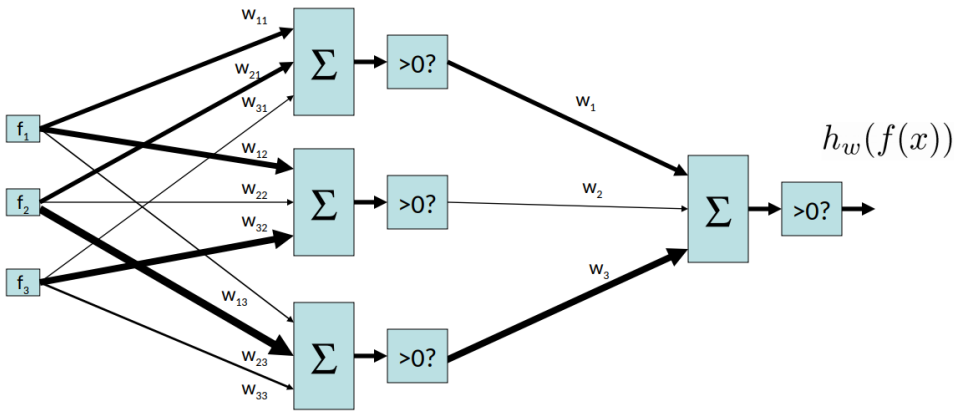

We would like to separate the two colors, and clearly there is no way this can be done in a single dimension (a single dimensional decision boundary would be a point, separating the axis into two regions).
To fix this problem, we can add additional (potentially nonlinear) features to construct a decision boundary from. Consider the same dataset with the addition of $x^2$ as a feature:



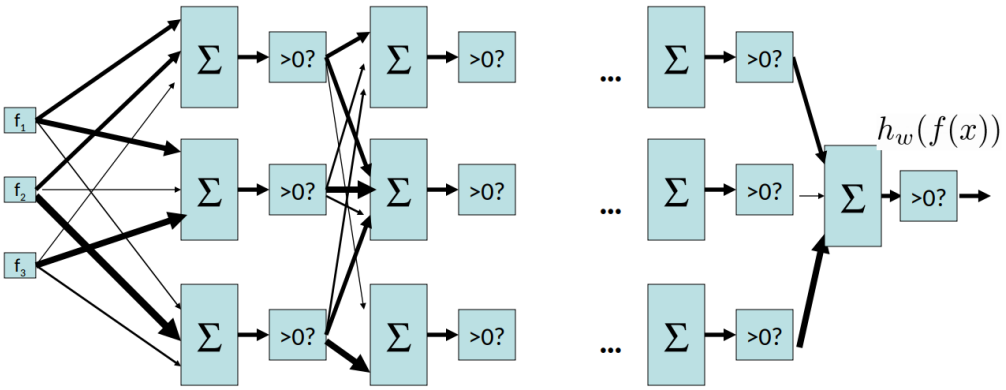

With this additional piece of information, we are now able to construct a linear separator in the two dimensional space containing the points. In this case, we were able to fix the problem by mapping our data to a higher dimensional space by manually adding useful features to datapoints. However, in many high-dimensional problems, such as image classification, manually selecting features that are useful is a tedious problem. This requires domain-specific effort and expertise, and works against the goal of generalization across tasks. A natural desire is to learn these featurization or transformation functions as well, perhaps using a nonlinear function class that is capable of representing a wider variety of functions.

## Multi-layer Perceptron

Let's examine how we can derive a more complex function from our original perceptron architecture. Consider the following setup, a two-layer perceptron, which is a perceptron that takes as input the outputs of another perceptron.



In fact, we can generalize this to an N-layer perceptron:




With this additional structure and weights, we can express a much wider set of functions. However, we run into the problem of selecting the best set of weights to parameterize our network.

# Measuring Accuracy

The accuracy of the binary perceptron can be expressed as:

$$l^{acc}(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m}(sgn(\boldsymbol{w}\cdot\mathbf{f}(x^{(i)})) == y^{(i)})$$

where $x^{(i)}$ is datapoint $i$, $\mathbf{f}$ is our function that derives a feature vector from a raw datapoint, and $y^{(i)}$ is the actual class label of the point.

Sometimes, we want an output that is more expressive than a binary label. It is sometimes useful to produce a probability for each class, reflecting our a degree of certainty that the datapoint belongs to that class. To do so, we can use the *softmax* function, which is defined as:

$$\sigma(\boldsymbol{z})_j = \frac{e^{z_j}}{\sum_{k=0}^{n} e^{z_k}} = P(Class\ j|\boldsymbol{z}) = P(Class\ j|f(x))$$

Given a vector that is output by our function $f$, softmax performs normalization to output a probability distribution. To come up with a general loss function for our models, we can use this probability distribution to generate an expression for the likelihood of a set of weights:

$$\ell(\boldsymbol{w}) = \prod_{i=1}^{m} P(y^{(i)}|f(x^{(i)});\boldsymbol{w})$$

This expression denotes the likelihood of a particular set of weights explaining the observed labels and datapoints. We would like to find the set of weights that maximizes this quantity. This is identical to finding the maximum of the log-likelihood expression (since log is an increasing function, the maximizer of one will be the maximizer of the other):

$$\ell\ell(\boldsymbol{w}) = \log\prod_{i=1}^{m} P(y^{(i)}|f(x^{(i)});\boldsymbol{w}) = \sum_{i=1}^{m}\log P(y^{(i)}|f(x^{(i)});\boldsymbol{w})$$

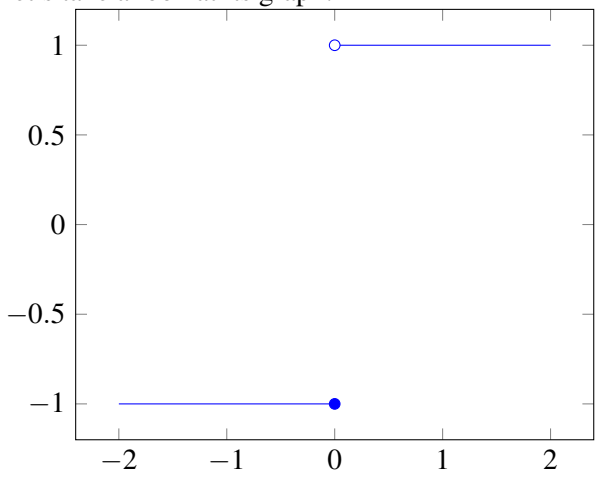
(Depending on the application, the formulation as a sum of log probabilities may be more useful – for example in mini-batched or stochastic gradient descent; see the Neural Networks: Optimization section below.) In the case where the log likelihood is differentiable with respect to the weights, we will discuss a simple algorithm to optimize it.

# Multi-layer Feedforward Neural Networks

We now introduce the idea of an (artificial) neural network. This is much like the multi-layer perceptron, however, we choose a different non-linearity to apply after the individual perceptron nodes. Remember that it is this non-linearity that makes the network as a whole non-linear (the composition of linear functions will also be linear). In the case of a multi-layer perceptron, we chose a step function:

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

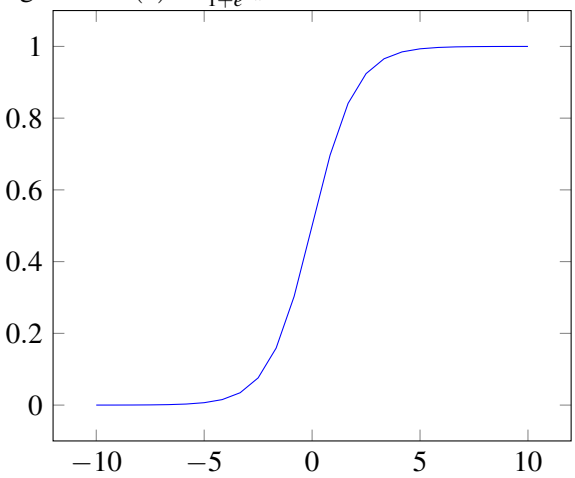
Let's take a look at its graph:



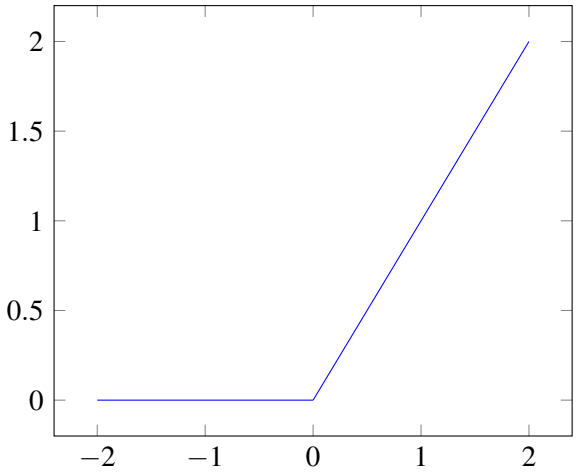

This is difficult to optimize for a number of reasons which will hopefully become clearer when we address gradient descent. Firstly, it is not continuous, and secondly, it has a derivative of zero at all points. Intuitively, this means that we cannot know in which direction to look for a local minima of the function, which makes it difficult to minimize loss in a smooth way.

So, instead of using a step function, we will use some continuous function. We have many options for such a function, including the sigmoid functions (named for the Greek $\sigma$ or 's' as it looks like an 's') as well as the "rectified linear unit" (ReLU) function.Let's look at their definitions/graphs below:

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




ReLU: $f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$




Calculating the output is done as before in the multi-layer perceptron, now applying one of our new non-linearities (chosen as part of the architecture for the neural network) instead of the old step function.

# Loss Functions and Multivariate Optimization

Now we have a sense of how a feed-forward neural network is constructed and makes its predictions, we would like to develop a way to train it, iteratively improving its accuracy, similarly to how we did in the case of the perceptron. In order to do so, we will need to be able to measure their performance. Returning to our log-likelihood function that we wanted to maximize, we can derive an intuitive algorithm to optimize our weights given that our function is differentiable.

### Gradient Ascent / Descent

To maximize our log-likelihood function, we differentiate it to obtain a *gradient* vector consisting of its partial derivatives for each parameter:

$$\nabla_w \ell\ell(\mathbf{w}) = \left[\frac{\partial \ell\ell(\mathbf{w})}{\partial \mathbf{w}_1}, \ldots, \frac{\partial \ell\ell(\mathbf{w})}{\partial \mathbf{w}_n}\right]$$

This gradient vector gives the local direction of steepest ascent (or descent if we reverse the vector). **Gradient ascent** is a greedy algorithm that calculates this gradient for the current values of the weight parameters, then updates the parameters along the direction of the gradient, scaled by a *step-size*, $\alpha$. Specifically the algorithm looks as follows:

Initialize weights $\boldsymbol{w}$
For i = 0, 1, 2, ...
  $\boldsymbol{w} = \boldsymbol{w} + \alpha \nabla_w \ell\ell(\boldsymbol{w})$

If rather than minimizing we instead wanted to minimize a function $f$, the update should subtract the scaled gradient ($\mathbf{w} = \mathbf{w} - \alpha \nabla_{\mathbf{w}} f(\mathbf{w})$) – this gives the **gradient descent** algorithm.

# Neural Networks: Backpropagation

To efficiently calculate the gradients for each parameter in a neural network, we will use the **backpropagation** algorithm. Backpropagation represents the neural network as a dependency graph, called a **com-**

**putational graph**. The graph structure will allow us to efficiently compute both the network's error (loss) on input data, as well as the gradients of each parameter with respect to the loss. We can then use these gradients in gradient descent to adjust the network's parameters and decrease the loss on the training data.

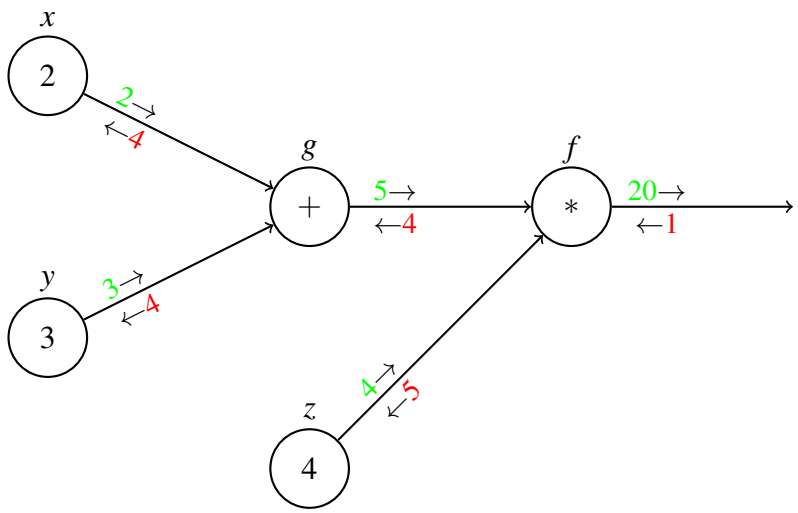



Figure 4: a computation graph for computing $(x+y)*z$ with the values $x=2$, $y=3$, $z=4$

Figure 4 shows an example computation graph for computing $(x+y)*z$ with the values $x=2$, $y=3$, $z=4$. We will write $g=x+y$ and $f=g*z$. Values in green are the outputs of each node, which we compute in the **forward pass**, where we apply each node's operation to its input values coming from its parent nodes.

Values in red after each node give gradients of the function computed by the graph, which are computed in the **backward pass**: the value after each node is the partial derivative of the last node, $f$'s, value with respect to the node's value. For example, the red value 4 after $g$ is $\frac{\partial f}{\partial g}$, and the red value 4 after $x$ is $\frac{\partial f}{\partial x}$. In our simple example, $f$ is just a multiplication node, but in a real neural network the final node will usually compute the loss value that we are trying to minimize.

The backward pass computes gradients by starting at the final node (which has a gradient of 1) and passing and updating gradients backward through the graph. Intuitively, each node's gradient measures how much a change in that node's value contributes to a change in the final node's value. This will be the product of how much the node contributes to a change in its child node, with how much the child node contributes to a change in the final node.

Each node receives and combines gradients from its children, updates this combined gradient based on the node's inputs and the node's operation, and then passes the updated gradient backward to its parents. Each node's gradient can be derived using the chain rule.

In the example in Figure 4, we want to find the gradients $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$, and $\frac{\partial f}{\partial z}$.

Since $f$ is our final node, it has gradient $\frac{\partial f}{\partial f}=1$. Then we compute the gradients for its parents, $g$ and $z$. We have $\frac{\partial f}{\partial g}=\frac{\partial}{\partial g}(g\cdot z)=z=4$, and $\frac{\partial f}{\partial z}=\frac{\partial}{\partial z}(g\cdot z)=g=5$.

Now we can move on upstream to compute the gradients of $x$ and $y$. For these, we'll use the chain rule and reuse the gradient we just computed for $g$, $\frac{\partial f}{\partial g}$. For $x$, we have $\frac{\partial f}{\partial x}=\frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$ by the chain rule – the product of the gradient coming from $g$ with the partial derivative for $x$ at this node. We have $\frac{\partial g}{\partial x}=\frac{\partial}{\partial x}(x+y)=\frac{\partial}{\partial x}x+\frac{\partial}{\partial x}y=0+1$, so $\frac{\partial f}{\partial x}=4\cdot 1$. Intuitively, the amount that a change in $x$ contributes to a change in $f$ is the product of the amount that a change in $g$ contributes to a change in $f$, with the amount that a change in

$x$ contributes to a change in $g$.

Since the backward pass step for a node in general depends on the node's inputs (which are computed in the forward pass), and gradients computed "downstream" of the current node by the node's children (computed earlier in the backward pass), we cache all of these values in the graph for efficiency. Taken together, the forward and backward pass over the graph make up the backpropagation algorithm.

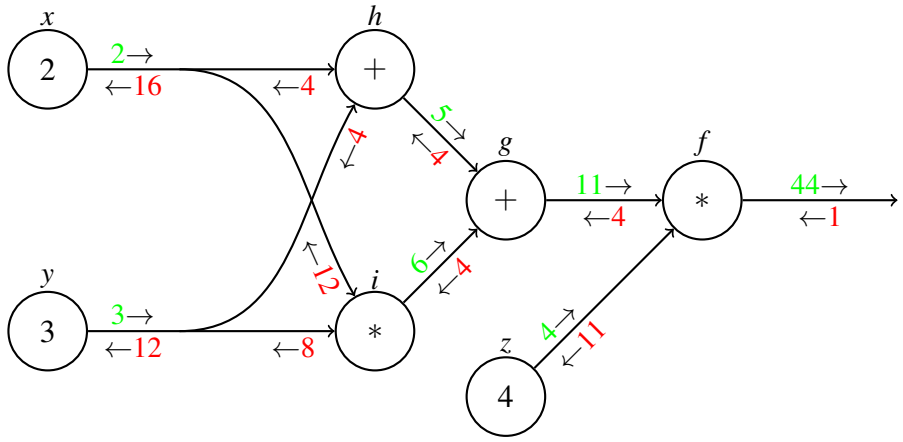



Figure 5: a computation graph for computing $((x+y)+(x\cdot y))\cdot z$, with $x=2$, $y=3$, $z=4$.

Nodes can also have multiple children, if they are used multiple times in computing the final value. For example, consider the graph in Figure 5, representing $((x+y)+(x\cdot y))\cdot z$, with $x=2$, $y=3$, $z=4$. $x$ and $y$ are each used in 2 operations, and so each has two children. By the multivariate chain rule, their gradient values are the sum of the gradients computed for them by their children (i.e. gradient values add at path junctions). For example, to compute the gradient for $x$, we have

$$\frac{\partial f}{\partial x}=\frac{\partial f}{\partial h}\frac{\partial h}{\partial x}+\frac{\partial f}{\partial i}\frac{\partial i}{\partial x}=4\cdot 1+4\cdot 3=4+12=16$$

# Neural Networks: Optimization

Now that we have a method for computing gradients for all parameters of the network, we can use gradient descent methods to optimize the parameters to get high accuracy on our training data. For example, suppose we have designed some classification network to output probabilities of classes $y$ for data points $x$, and have $m$ different training datapoints (see the "Measuring Accuracy" section for more on this). Let $\mathbf{w}$ be all the parameters of our network. We want to find values for the parameters $\mathbf{w}$ that maximize the likelihood of the true class probabilities for our data, so we have the following function to run gradient ascent on:

$$\ell\ell(\mathbf{w})=\log\prod_{i=1}^{m}(P(y^{(i)}\mid f(x^{(i)});\mathbf{w})=\sum_{i=1}^{m}\log(P(y^{(i)}\mid f(x^{(i)});\mathbf{w})$$

where $x^{(1)}\ldots x^{(m)}$ are the $m$ datapoints in our training set.

One way to try to minimize this function is, at each iteration of gradient descent, to use all the data points $1\ldots m$ to compute gradients for the parameters $\mathbf{w}$, update the parameters, and repeat until the parameters converge (at which point we've reached a local minimum of the function).

However, this is rarely done in practice, since datasets are typically large enough that computing gradients for this full likelihood function will be very slow. Instead, we'll typically use **mini-batching**. Mini-batching

rotates through *batches* of $k$ data points at a time, taking one batch for each step of gradient descent and computing gradients of the loss function using only that batch (so that the sum above is over the $k$ datapoints in the batch, rather than all $m$ datapoints in the training set). This allows us to compute each gradient update much more quickly, and often still makes fast progress toward the minimum of the function. The limit where the batch size $k = 1$ is known as **stochastic gradient descent (SGD)**. In SGD, we randomly sample a single example from the training dataset at each step of gradient descent, compute parameter gradients using the network's loss on that single example, update the parameters, and repeat (sampling another example from the training set).

Neural networks are powerful (and universal!) function approximators, but can be difficult to design and train. There's a lot of ongoing research on architectures (designing a net that's a good fit for a particular problem), learning (how to find parameters that achieve a low value of the loss function – difficult since gradient descent is a greedy algorithm, and neural nets can have many local optima), and generalization (since neural nets have many parameters, it's often easy to overfit training data – how do you guarantee that they also have low loss on testing data you haven't seen before?), among other topics. If you're interested, we hope you continue on in a class like cs189 to explore these areas more.