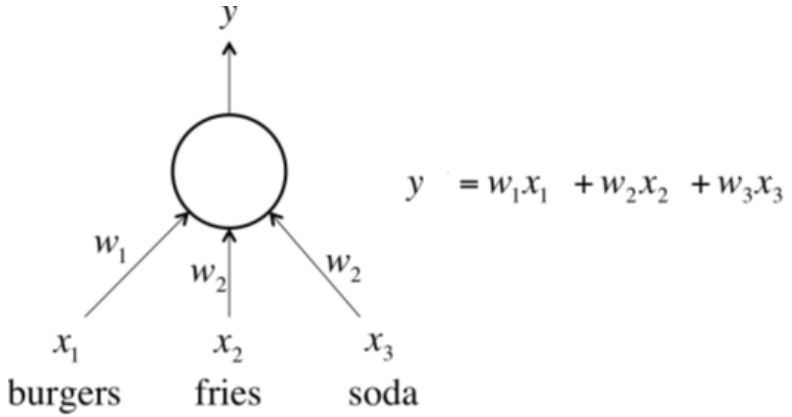
Unit 3
Training Feed Forward Neural Networks

Prepared By: Arjun Singh Saud
Asst. Prof. CDCSIT

The Fast Food Problem

- Fast food problem is the problem of determining price of items purchased in a fast-food restaurant.
- In the previous chapter we have devised a perceptron for solving this problem where input variables are number of items purchased and weights are cost of the item.



The Fast Food Problem

- Suppose we don't know price of items purchased. But we have purchase history data that contains quantities of each item purchased and total price and we want to predict total price of purchase made in a particular day.
- For this, we have to train the neuron so that we pick the optimal weights possible-the weights that minimize the errors we make on the training examples.
- In this case, let's say we want to minimize the square error over all of the training examples that we encounter.

The Fast Food Problem

- More formally, if we know that t_i is the true answer for the i^{th} training example and y_i is the value computed by the neural network, we want to minimize the value of the error function E.

$$E = \frac{1}{2}\sum_{i=0}^{n}(t_i - y_i)^2$$

- The squared error is zero when our model makes a perfectly correct prediction on every training example. Moreover, the closer E is to 0, the better our model is. As a result, our goal will be to select our parameter vector θ (the values for all the weights in our model) such that E is as close to 0 as possible.

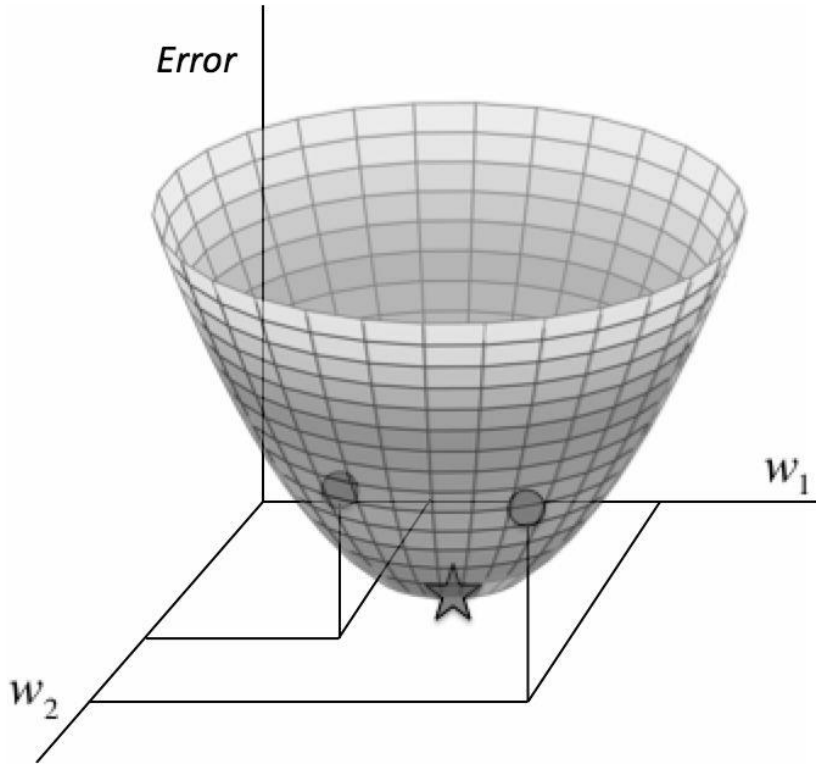
The Gradient Descent

- Gradient descent is an optimization algorithm used to minimize some convex function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient.
- In machine learning, we use gradient descent to update the parameters of our model. Parameters refer to coefficients in Logistic Regression and weights in neural networks.

The Gradient Descent

- Let's say our linear neuron only has two inputs (and thus only two weights, w_1 and w_2). Then we can imagine a three-dimensional space where the horizontal dimensions correspond to the weights w_1 and w_2, and the vertical dimension corresponds to the value of the error function E.
- If we consider the errors we make over all possible weights, we get a surface in this three-dimensional space, in particular, a quadratic bowl as shown in the Figure of next slide.

The Gradient Descent



The Gradient Descent

- We can also conveniently visualize this surface as a set of elliptical contours, where the minimum error is at the center of the ellipses.
- In this setup, we are working in a two-dimensional plane where the dimensions correspond to the two weights. The closer the contours are to each other, the steeper the slope.
- In fact, it turns out that the direction of the steepest descent is always perpendicular to the contours. This direction is expressed as a vector known as the *gradient*.

The Gradient Descent

- Suppose we randomly initialize the weights of our network so we find ourselves somewhere on the horizontal plane.
- By evaluating the gradient at that position, we can find the direction of steepest descent, and we can take a step in that direction. This will takes us to a new position that's closer to the minimum.
- We can reevaluate the direction of steepest descent by taking the gradient at this new position and taking a step in this new direction as shown in the Figure in the next slide.
- Following this strategy will eventually takes us to the point of minimum error.

The Gradient Descent

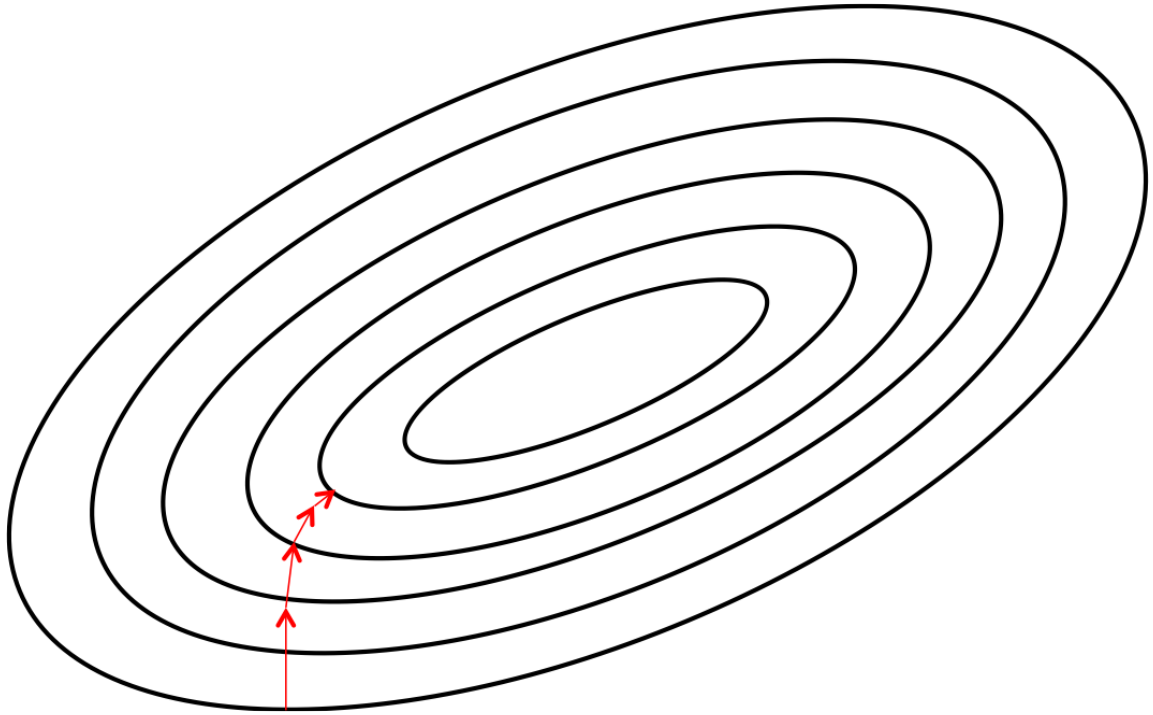


Figure: Visualizing the error surface as a set of contours

The Gradient Descent

- Mathematically, if E is function to be minimized (cost function), Gradient descent changes the parameters of learning model iteratively as below:

$$w = w - \alpha \frac{df}{dw}$$

where, α is learning rate and w is parameter to be optimized

The Delta Rule and Learning Rates

- Learning rate is a hyperparameter that needs to be set properly for proper and efficient working of learning algorithms. Value of learning rate ranges between 0-1.
- Hyperparameters are the parameters whose value can not be learned from training data.
- How much bigger steps we take towards minima is determined from two factors: size of gradient and learning rate.
- If we pick a learning rate that's too small, we risk taking too long during the training process. But if we pick a learning rate that's too big, we'll mostly likely start diverging away from the minimum.

The Delta Rule and Learning Rates

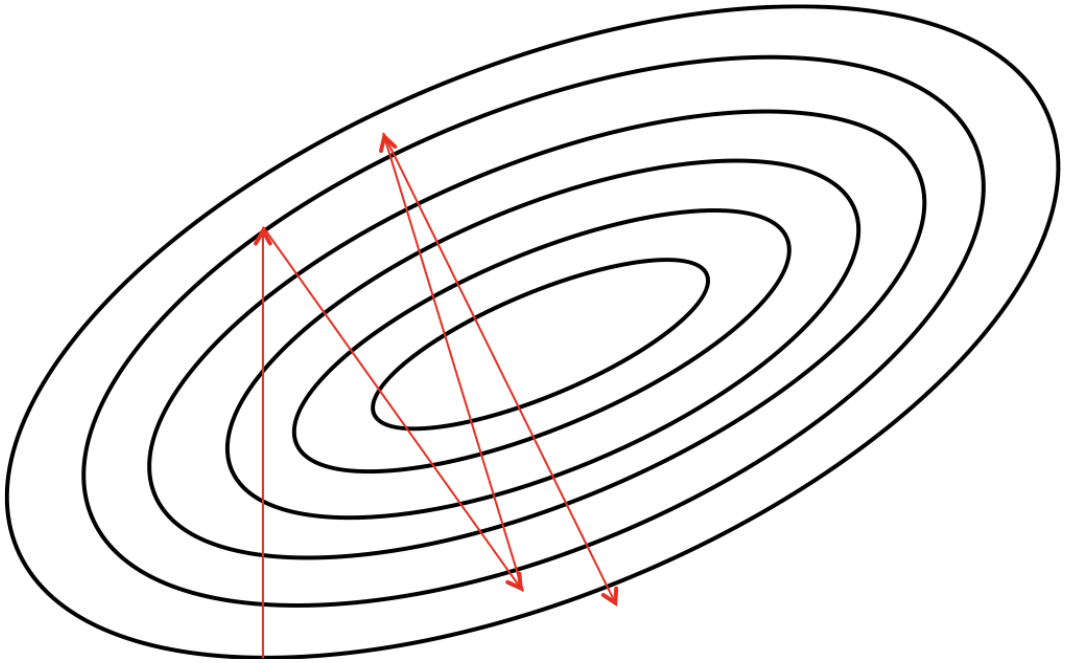


Figure: Convergence is difficult when learning rate is too large

The Delta Rule and Learning Rates

- Delta rule is a method of training perceptron on the basis of gradient of error function.
- In order to calculate how to change each weight, we evaluate the gradient, which is essentially the partial derivative of the error function with respect to each of the weights.

$$\Delta w_k = -\alpha \frac{\partial E}{\partial w_k} = -\alpha \frac{\partial\left(\frac{1}{2}\sum_{i=0}^{n}(t_i - y_i)^2\right)}{\partial w_k}$$

$$= -\alpha \frac{\partial\left(\frac{1}{2}\sum_{i=0}^{n}(t_i - y_i)^2\right)}{\partial y_i} \frac{\partial y_i}{\partial w_k} = \alpha \sum_{i=1}^{n}(t_i - y_i)\, x_k^{(i)}$$

Gradient Descent with Sigmoidal Neurons

- Let's recall the mechanism by which logistic neurons compute their output value from their inputs.

$$z = \sum_k x_k w_k$$

$$y = \frac{1}{1+e^{-z}}$$

- The neuron computes the weighted sum of its inputs, z. It then feeds this net input, into the input function to compute y, its final output.
- We want to compute the gradient of the error function with respect to the weights.

Gradient Descent with Sigmoidal Neurons

- We want to compute the gradient of the error function with respect to the weights.

$$\frac{\partial E}{\partial w_k} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial z_i}\frac{\partial z_i}{\partial w_k}$$

$$\frac{\partial E}{\partial y_i} = -\sum_{i=1}^{n}(t_i - y_i)$$

$$\frac{\partial y_i}{\partial z_i} = \frac{\partial(1+e^{-z_i})^{-1}}{\partial z_i} = e^{-z_i}(1+e^{-z_i})^{-2} = \frac{e^{-z_i}}{(1+e^{-z_i})^2} = \frac{1+e^{-z_i}-1}{(1+e^{-z_i})^2} = \frac{1+e^{-z_i}}{(1+e^{-z_i})^2} - \frac{1}{(1+e^{-z_i})^2}\, y_i - y_i^2 = y_i(1-y_i)$$

$$\frac{\partial z_i}{\partial w_k} = x_k^{(i)}$$

=> $\Delta w_k = \alpha \sum_{i=1}^{n}(t_i - y_i)\, y_i(1-y_i)x_k^{(i)}$

Gradient Descent with Sigmoidal Neurons

- If we consider general sigmoidal neurons to compute their output value from their inputs.

$$z = \sum_k x_k w_k$$

$$y = \frac{1}{1+e^{-az}}$$

Gradient Descent with Sigmoidal Neurons

- We want to compute the gradient of the error function with respect to the weights.

$$\frac{\partial E}{\partial w_k} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial z_i}\frac{\partial z_i}{\partial w_k}$$

$$\frac{\partial E}{\partial y_i} = -\sum_{i=1}^{n}(t_i - y_i)$$

$$\frac{\partial y_i}{\partial z_i} = \frac{\partial(1+e^{-az_i})^{-1}}{\partial z_i} = a\, e^{-az_i}(1+e^{-az_i})^{-2} = \frac{ae^{-az_i}}{(1+e^{-az_i})^2} = \frac{a(1+e^{-az_i}-1)}{(1+e^{-z_i})^2} = a(y_i - y_i^2) = ay_i(1-y_i)$$

$$\frac{\partial z_i}{\partial w_k} = x_k^{(i)}$$

=> $\Delta w_k = \alpha \sum_{i=1}^{n}(t_i - y_i)\, a\, y_i(1-y_i)x_k^{(i)}$

Gradient Descent with Sigmoidal Neurons

- If we consider tanh neurons to compute their output value from their inputs.

$$z = \sum_k x_k w_k$$

$$y = tanhz$$

Gradient Descent with Sigmoidal Neurons

- We want to compute the gradient of the error function with respect to the weights.

$$\frac{\partial E}{\partial w_k} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial z_i}\frac{\partial z_i}{\partial w_k}$$

$$\frac{\partial E}{\partial y_i} = -\sum_{i=1}^{n}(t_i - y_i)$$

$$\frac{\partial y_i}{\partial z_i} = \frac{\partial tanhz_i}{\partial z_i} = \frac{\partial}{\partial z_i}\frac{sinhz_i}{coshz_i} = \frac{coshz_i \frac{\partial}{\partial z_i} sinhz_i - sinhz_i \frac{\partial}{\partial z_i} coshz_i}{(cosh)^2 z_i} = \frac{(cosh)^2 z_i - (sinh)^2 z_i}{(cosh)^2 z_i}$$

$$= 1 - \frac{(cosh)^2 z_i}{(cosh)^2 z_i} = 1 - (tanh)^2 z_i = 1 - y^2 = (1-y) + (1-y)$$

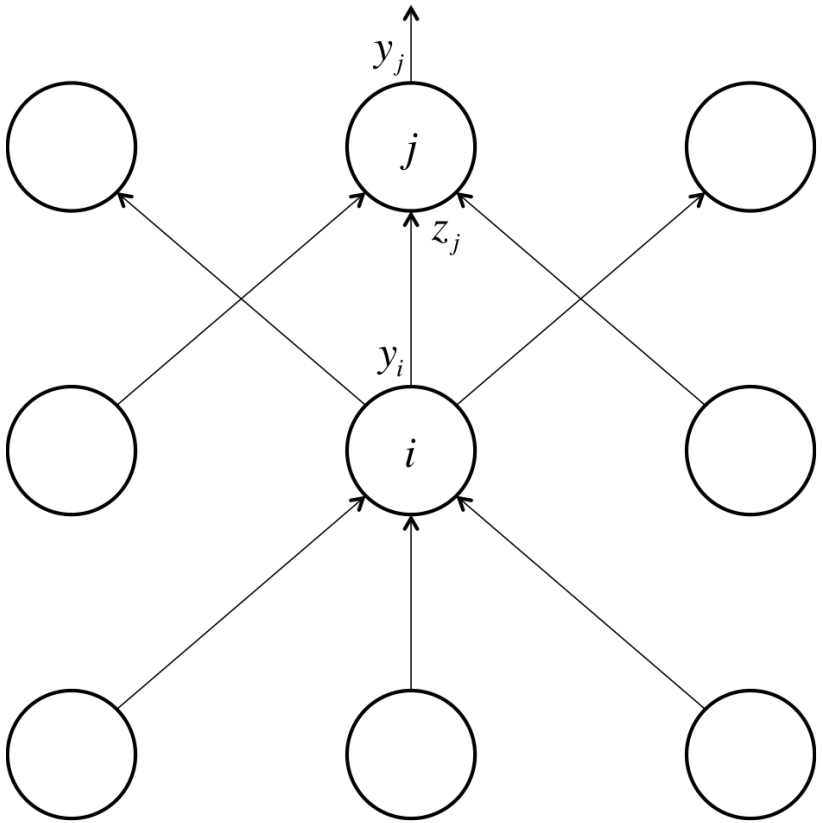
$$\frac{\partial z_i}{\partial w_k} = x_k^{(i)}$$

=> $\Delta w_k = \alpha \sum_{i=1}^{n}(t_i - y_i)\,(1 + y_i)(1 - y_i)x_k^{(i)}$

The Backpropagation Algorithm

- Backpropagation is the most popular method for the training of multilayer perceptron. The training proceeds in two phases:
  - **Forward Phase**: In this phase, input signal is propagated through the network, layer by layer, until it reaches the output layer and output of each neuron is calculated.
  - **Backward Phase**: In this phase, an error signal is produced by comparing the output of the network with a desired response. Gradient of the error is calculated and the resulting gradient is propagated in backward direction through the network, again layer by layer, until it reaches the first hidden layer. In this phase, adjustments are made to the synaptic weights of the network.

The Backpropagation Algorithm



Reference diagram for the derivation of the Backpropagation algorithm

The Backpropagation Algorithm

- Let us assume that the subscript refer to the layer of the neuron, the symbol y refer to the activity or outcome of a neuron, the symbol z refer to the of the net input of the neuron, and w_{ij} denote the weight of link between neuron i and neuron j.
- Thus, for each neuron in the network net input and output is calculated as below. (Assuming Logistic Activation Function)

$$z_j = \sum w_{ij} y_i$$

$$y_j = \frac{1}{1+e^{-z_i}}$$

The Backpropagation Algorithm

- Now, error function at output layer j of the network is given as below.

$$E_j = \frac{1}{2}(t_j - y_j)^2$$

- Thus, gradient of error function w.r.t. weight w_{ij} at output layer j of the network is given as below.

$$\frac{\partial E_j}{\partial w_{ij}} = \frac{\partial E_j}{\partial y_j}\frac{\partial y_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}} = -(t_j - y_j)y_j(1 - y_j)y_i = -\delta_j y_i$$

where $\delta_j = (t_j - y_j)y_j(1 - y_j)$

The Backpropagation Algorithm

- It is not possible to compute error gradient at hidden layer neurons of network directly because desired outcome of the hidden layer neurons is unknown.
- Thus, we need to compute error gradient of neurons at hidden layer j in terms of error gradient of next layer neurons are as below.

$$\frac{\partial E_j}{\partial w_{ij}} = \frac{\partial E_k}{\partial y_k}\frac{\partial y_k}{\partial z_k}\frac{\partial z_k}{\partial y_j}\frac{\partial y_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}}$$

$$= -(t_k - y_k)y_k(1 - y_k)w_{jk}y_j(1 - y_j)y_i = -\delta_k w_{jk}y_j(1 - y_j)y_i$$

The Backpropagation Algorithm

- There can be any number of neurons in next layer (i.e. layer k) that are affected by neuron in layer j. Thus, error gradient for neuron in n layer can be calculated as below.

$$\frac{\partial E_j}{\partial w_{ij}} = -y_j(1-y_j)y_i \sum_k \delta_k w_{jk} = -\delta_j y_i,$$

Where, $\delta_j = y_j(1-y_j)\sum_k \delta_k w_{jk}$

- Finally, the correction applied to the weight and bias is defined as below.

$$\Delta w_{ij} = -\alpha \frac{\partial E_j}{\partial w_{ij}} = \alpha \delta_j y_i$$

The Backpropagation Algorithm

Algorithm

1. Initialize all weights and biases in *network*
2. Repeat until terminating condition is not satisfied
3. for each training tuple $\boldsymbol{x}$ in D *//Forward Pass*
   - for each input layer unit j

     $y_j = x_j$;
   - for each hidden or output layer unit j

     compute the net input of unit j

     $z_j = \sum_i w_{ji} y_i$

     compute the output of each unit j

     $y_j = 1/(1+e^{-z_j})$

The Backpropagation Algorithm

4. for each unit j in the output layer compute error gradient as below(Backward Pass Steps 4-6)

$$\delta_j = y_j(1 - y_j)(t_j - y_j)$$

5. for each unit j in the hidden layers error gradient as below

$$\delta_j = y_j(1 - y_j)\sum_k w_{jk}\,\delta_k$$

6. for each weight w_{ji} in network, update weights as below

$$w_{ji} = w_{ji} + \alpha\delta_j y_i$$

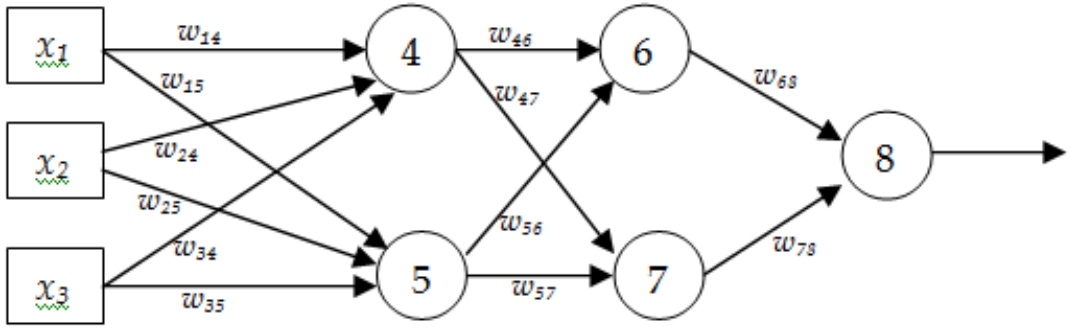
7. for each bias b_j in network, update biases as below

$$b_j = b_j + \alpha\delta_j$$

The Backpropagation Algorithm

Example

- Consider a MLP given below. Let the learning rate be 1. The initial weights of the network are given in the table below. Assume that first training tuple is (1, 0, 1) and its target output is 1. Calculate weight updates by using back-propagation algorithm. Assume $\varphi(x) = \frac{1}{1+e^{-x}}$.



w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{47}	w_{56}	w_{57}	w_{68}	w_{78}
0.6	0.4	0.2	-0.3	0.7	-0.6	0.4	0.7	0.1	0.8	0.2	0.5

The Backpropagation Algorithm

Solution: *Forward Pass*

$z_4=1*0.6+0*0.2+1*0.7=1.3$ $y_4=1/1+e^{-1.3}=0.786$

$z_5=1*0.4+0*(-0.3)+1*(-0.6)=-0.2$ $y_5=1/1+e^{0.2}=0.45$

$z_6=0.786*0.4+0.45*0.1=0.36$ $y_6=1/1+e^{-0.36}=0.59$

$z_7=0.786*0.7+0.45*0.8=0.91$ $y_7=1/1+e^{-0.91}=0.71$

$z_8=0.59*0.2+0.71*0.5=0.47$ $y_8=1/1+e^{-0.47}=0.61$

The Backpropagation Algorithm

Solution: *Backward Pass*

$$\delta_8 = y_8(1-y_8)(t_8-y_8) = 0.61*(1-0.61)*(1-0.61) = 0.093$$

$$\delta_7 = y_7(1-y_7)\delta_8 w_{78} = 0.71*(1-0.71)*0.093*0.5 = 0.0096$$

$$\delta_6 = y_6(1-y_6)\delta_8 w_{68} = 0.59*(1-0.59)*0.093*0.2 = 0.0045$$

$$\delta_5 = y_5(1-y_5)(\delta_7 w_{57} + \delta_6 w_{56}) = 0.002$$

$$\delta_4 = y_4(1-y_4)(\delta_7 w_{47} + \delta_6 w_{46}) = 0.0014$$

The Backpropagation Algorithm

Update Weights

$$w_{14} = w_{14} + 1 * \delta_4 * y_1 = 0.6 + 1 * 0.0014 * 1 = ?$$

$$w_{15} = w_{15} + 1 * \delta_5 * y_1 = ? \qquad w_{24} = w_{24} + 1 * \delta_4 * y_2 = ?$$

$$w_{25} = w_{25} + 1 * \delta_5 * y_2 = ? \qquad w_{34} = w_{34} + 1 * \delta_4 * y_3 = ?$$

$$w_{35} = w_{35} + 1 * \delta_5 * y_3 = ? \qquad w_{46} = w_{46} + 1 * \delta_6 * y_4 = ?$$

$$w_{47} = w_{47} + 1 * \delta_7 * y_4 = ? \qquad w_{56} = w_{56} + 1 * \delta_6 * y_5 = ?$$

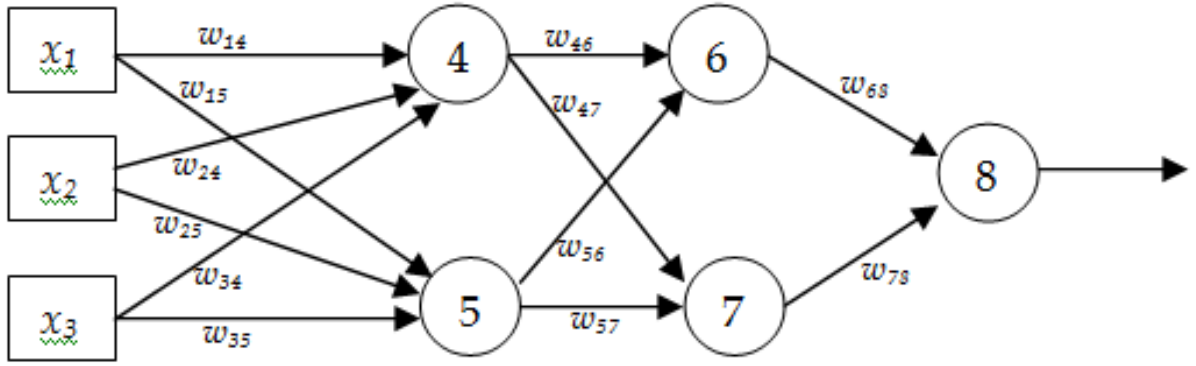
$$w_{57} = w_{57} + 1 * \delta_7 * y_5 = ? \qquad w_{68} = w_{68} + 1 * \delta_8 * y_6 = ?$$

$$w_{78} = w_{78} + 1 * \delta_8 * y_7 = ?$$

The Backpropagation Algorithm

HW

- Consider a MLP given below. Let the learning rate be 1. The initial weights of the network are given in the table below. Assume that first training tuple is (1, 0, 1) and its target output is 1. Calculate weight updates by using back-propagation algorithm. Assume $\varphi(x) = \tanh$.



w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{47}	w_{56}	w_{57}	w_{68}	w_{78}
0.6	0.4	0.2	-0.3	0.7	-0.6	0.4	0.7	0.1	0.8	0.2	0.5

The Backpropagation Algorithm

Derivation using Binary Cross Entropy

- In this case, error function at output layer j of the network is given as below.

$$E_j = -\{(1 - t_j)\log(1 - y_j) + t_j \log y_j\}$$

- Thus, gradient of error function w.r.t. weight w_{ij} at output layer j of the network is given as below.

$$\frac{\partial E_j}{\partial w_{ij}} = \frac{\partial E_j}{\partial y_j}\frac{\partial y_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}}$$

The Backpropagation Algorithm

$$\frac{\partial E_j}{\partial y_j} = \frac{1-t_j}{1-y_j} - \frac{t_j}{y_j}$$

$$\frac{\partial E_j}{\partial w_{ij}} = \frac{\partial E_j}{\partial y_j}\frac{\partial y_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}} = \left(\frac{1-t_j}{1-y_j} - \frac{t_j}{y_j}\right) y_j(1-y_j) y_i = \delta_j y_i$$

where $\delta_j = \left(\frac{1-t_j}{1-y_j} - \frac{t_j}{y_j}\right) y_j(1-y_j)$

The Backpropagation Algorithm

- It is not possible to compute error gradient at hidden layer neurons of network directly because desired outcome of the hidden layer neurons is unknown.
- Thus, we need to compute error gradient of neurons at hidden layer j in terms of error gradient of next layer neurons are as below.

$$\frac{\partial E_j}{\partial w_{ij}} = \frac{\partial E_k}{\partial y_k}\frac{\partial y_k}{\partial z_k}\frac{\partial z_k}{\partial y_j}\frac{\partial y_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}}$$

$$= \left(\frac{1-t_k}{1-y_k} - \frac{t_k}{y_k}\right) y_k(1-y_k)w_{jk}y_j(1-y_j)y_i = \delta_k w_{jk} y_j(1-y_j)y_i$$

The Backpropagation Algorithm

- There can be any number of neurons in next layer (i.e. layer k) that are affected by neuron in layer j. Thus, error gradient for neuron in n layer j can be calculated as below.

$$\frac{\partial E_j}{\partial w_{ij}} = y_j(1 - y_j)y_i \sum_k \delta_k w_{jk} = \delta_j y_i,$$

Where, $\delta_j = y_j(1 - y_j) \sum_k \delta_k w_{jk}$

- Finally, the correction applied to the weight and bias is defined as below.

$$\Delta w_{ij} = -\alpha \frac{\partial E_j}{\partial w_{ij}} = -\alpha \delta_j y_i$$

Gradient Descent Variation

- The three main flavors of gradient descent are *batch, stochastic, and mini-batch*.
- Batch gradient descent, computes the gradient of the cost function w.r.t. the parameters w for the entire training dataset:

$$w = w - \alpha \frac{df(w, x, y)}{dw}$$

- As we need to calculate the gradients for the whole dataset perform just *one* update, batch gradient descent is computationally fast. However, it is intractable for datasets that don't fit in memory.

Gradient Descent Variation

- Batch gradient descent also doesn't allow us to update our model *online*, i.e. with new examples on-the-fly. Pseudocode of batch gradient descent looks like below:

  for i in range(#epochs):

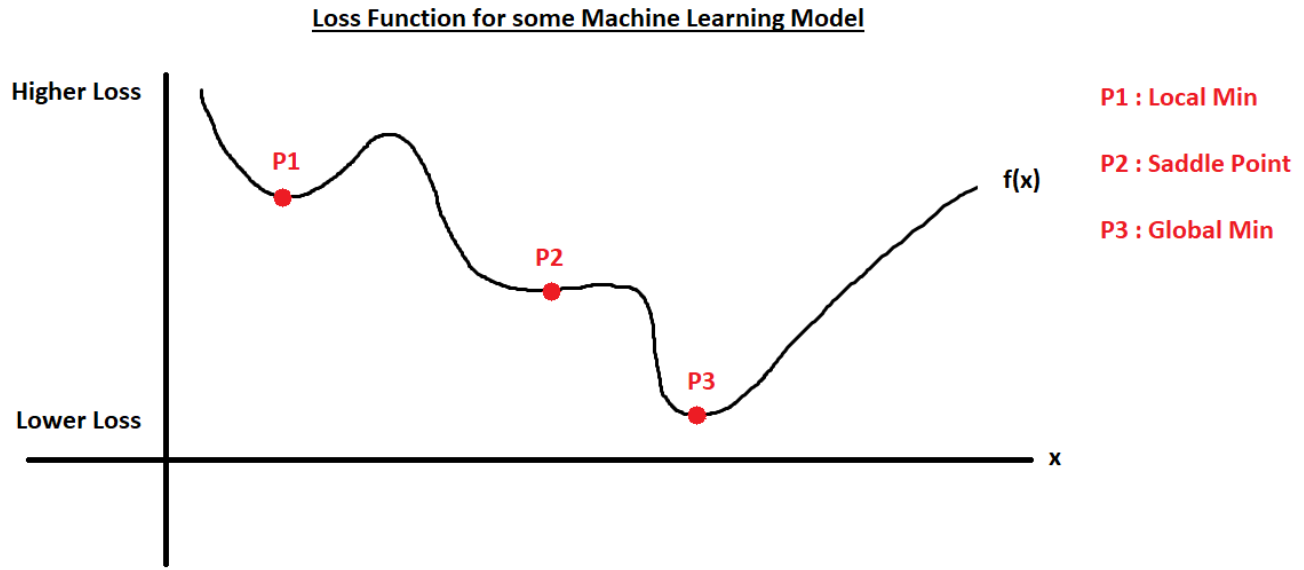
  grad = evaluategradient(data, para)

  *para= para – learning_rate * grad*

- Batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces. However, if the error surface is non-convex, it may converge in local minima.

Gradient Descent Variation

- Additionally, The error surface may have a flat region (also known as saddle point in high-dimensional spaces), and batch gradient descent may stuck may stuck in saddle points and local minimum while performing gradient descent.



Gradient Descent Variation

- Stochastic gradient descent (SGD) in contrast performs a parameter update for each training example $x^{(i)}$ and label $y^{(i)}$. Therefore, learning happens on every example.

$$w = w - \alpha \frac{df(w, x^{(i)}, y^{(i)})}{dw}$$

- The term *stochastic* indicates that the one example comprising each batch is chosen at random.
- Stochastic gradient descent strategy allow us to update our model *online*.

Gradient Descent Variation

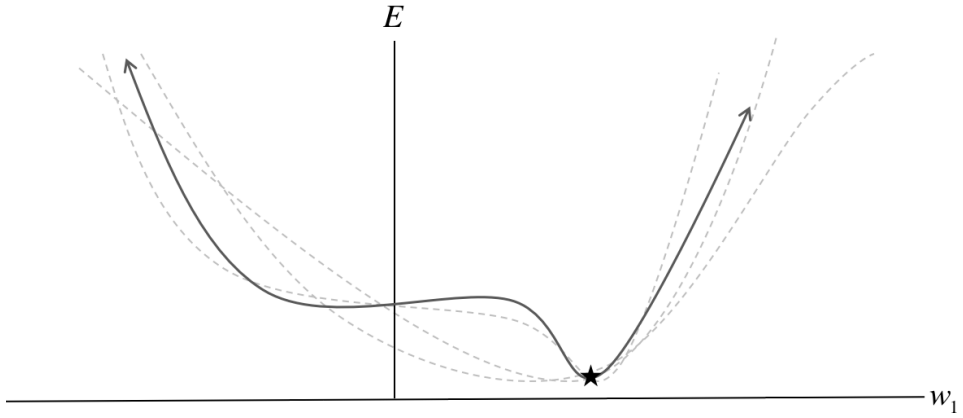
- Pseodocode for Stochastic Gradient Descent looks like below:

```
for i in range(#epochs):
        np.random.shuffle(data)
        for d in data:
                grad = compute_gradient(d, params)
                params = params – learning_rate * grad
```

- SGD updates the model much frequently, which is more computationally expensive than other batch gradient descent.
- Thus it takes significantly longer to train models on large datasets. At the same time we lose speedup due to Vectorization.

Gradient Descent Variation

- In SGD, instead of a single static error surface, our error surface is dynamic.



- Thus, these frequent updates can result in a noisy gradient signal, which may cause the model parameters jump around (have a higher variance over training epochs).

Gradient Descent Variation

- As a result, descending on this stochastic surface significantly improves our ability to navigate flat regions and local minimum.
- Additionally, the stochastic behavior complicates convergence to the global minimum.
- One way to combat this problem is using mini-batch gradient descent. In mini-batch gradient descent, at every iteration, we compute the error surface with respect to some subset of the total dataset (instead of just a single example). This subset is called a minibatch.

Gradient Descent Variation

$$w = w - \alpha \frac{df\left(w, x^{(i:i+n)}, y^{(i:i+n)}\right)}{dw}$$

- Pseudocode of Mini-batch gradient descent looks like below:

```
for i in range(#epochs):
        np.random.shuffle(data)
        for batch in data:
                grad = compute_gradient(batch, params)
                params = params  −  learning_rate * grad
```

Gradient Descent Variation

- Mini-batch gradient descent strike a balance between the efficiency of batch gradient descent and the local-minima avoidance afforded by stochastic gradient descent.

Training Sets, Validation Sets, and Overfitting

- One of the major issues with artificial neural networks is that the models are quite complicated.
- For example, let's consider a neural network that's pulling data from an image from the MNIST database (28 x 28 pixels), feeds into two hidden layers with 30 neurons, and finally reaches a softmax layer of 10 neurons. The total number of parameters in the network is nearly 25,000.
- By building a very complex model, it's quite easy to perfectly fit our training dataset because we give our model enough degrees of freedom to contort itself to fit the observations in the training set.

Training Sets, Validation Sets, and Overfitting

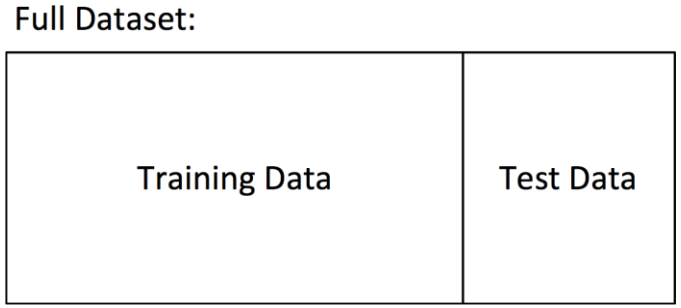
- But when we evaluate such a complex model on new data, it performs very poorly. In other words, the model does not generalize well. This is a phenomenon called overfitting, and it is one of the biggest challenges that a machine learning engineer must combat.
- This becomes an even more significant issue in deep learning, where our neural networks have large numbers of layers containing many neurons. The number of connections in these models is astronomical, reaching the millions. As a result, overfitting is commonplace.

Training Sets, Validation Sets, and Overfitting

- This leads to three major observations. First, the machine learning engineer is always working with a direct trade-off between overfitting and model complexity.
- If the model isn't complex enough, it may not be powerful enough to capture all of the useful information necessary to solve a problem.
- However, if our model is very complex (especially if we have a limited amount of data), we run the risk of overfitting.
- Thus, deep learning takes the approach of solving very complex problems with complex models and taking additional countermeasures to prevent overfitting.

Training Sets, Validation Sets, and Overfitting

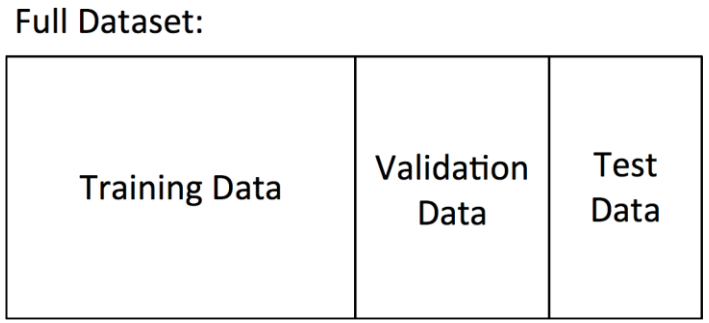
- Second, it is very misleading to evaluate a model using the data we used to train it. This would falsely suggest us to use complex model is preferable to a linear simple model.
- As a result, we almost never train our model on the entire dataset. Instead, we split up our data into a training set and a test set.



- This enables us to make a fair evaluation of our model by directly measuring how well it generalizes on new data it has not yet seen

Training Sets, Validation Sets, and Overfitting

- Third, it's quite likely that while we're training our data, there's a point in time where instead of learning useful features, we start overfitting to the training set. To avoid that, we want to be able to stop the training process as soon as we start overfitting.
- To do this, we divide our training process into *epochs*. At the end of each epoch, we want to measure how well our model is generalizing. To do this, we use an additional *validation set*.



Training Sets, Validation Sets, and Overfitting

- At the end of an epoch, the validation set will tell us how the model does on data it has yet to see. If the accuracy on the training set continues to increase while the accuracy on the validation set stays the same (or decreases), it's a good sign that it's time to stop training because we're overfitting.
- The validation set is also helpful as a proxy measure of accuracy during the process of *hyperparameter optimization*.
- One potential way to find the optimal setting of hyperparameters is by applying a *grid search*.

Training Sets, Validation Sets, and Overfitting

- In grid search, we pick a value for each hyperparameter from a finite set of options (e.g., $\alpha \in \{0 .001, 0.01, 0.1\ldots\}$, batch size $\in$ {16, 64, 128 , ...}, and train the model with every
- possible permutation of hyperparameter choices. We elect the combination of hyperparameters with the best performance on the validation set, and report the accuracy of the model trained with best combination on the test set

Preventing Overfitting in Deep Neural Networks

- There are several techniques that have been proposed to prevent overfitting during the training process. In this section, we'll discuss these techniques in detail.
- One method of combatting overfitting is called *regularization*. Regularization modifies the objective function that we minimize by adding additional terms that penalize large weights.
- In other words, we change the objective function so that it becomes $Error + \lambda\ f(\theta)$, where $f(\theta)$ grows larger as the components of θ grow larger, and λ is the regularization strength (hyperparameter).

Preventing Overfitting in Deep Neural Networks

- The value we choose for λ determines how much we want to protect against overfitting. A $\lambda = 0$ implies that we do not take any measures against the possibility of overfitting.
- If λ is too large, then our model will prioritize keeping θ as small as possible over trying to find the parameter values that perform well on our training set.
- As a result, choosing λ is a very important task and can require some trial and error.

Preventing Overfitting in Deep Neural Networks

- The most common type of regularization in machine learning is *L_2 regularization*. It can be implemented by augmenting the error function with the squared magnitude of all weights in the neural network.
- In other words, for every weight w in the neural network, we add $\frac{1}{2}\lambda w^2$ to the error function.
- The L_2 regularization has the intuitive interpretation of heavily penalizing peaky weight vectors and preferring diffuse weight vectors.

Preventing Overfitting in Deep Neural Networks

- This has the appealing property of encouraging the network to use all of its inputs a little rather than using only some of its inputs a lot.
- Gradient descent update using the L_2 regularization ultimately means that every weight is decayed linearly to zero.
- Because of this phenomenon, L_2 regularization is also commonly referred to as *weight decay*.

Preventing Overfitting in Deep Neural Networks

- Another common type of regularization is L_1 *regularization*. Here, we add the term λw for every weight w in the neural network.
- The L_1 regularization has the interesting property that it leads the weight vectors to become sparse during optimization (i.e., very close to exactly zero).
- In other words, neurons with L1 regularization end up using only a small subset of their most important inputs and become quite resistant to noise in the inputs.

Preventing Overfitting in Deep Neural Networks

- In comparison, weight vectors from L_2 regularization are usually diffuse, small numbers. L_1 regularization is very useful when you want to understand exactly which features are contributing to a decision.
- If this level of feature analysis isn't necessary, we prefer to use L_2 regularization because it empirically performs better.

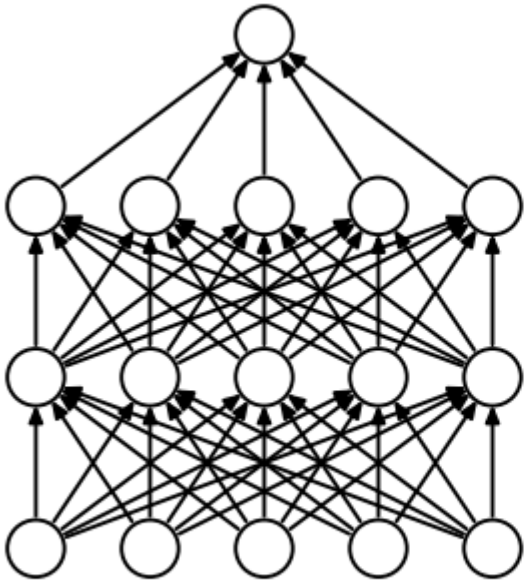
Preventing Overfitting in Deep Neural Networks

- *Max norm constraints* have a similar goal of attempting to restrict θ from becoming too large, but they do this more directly.
- Max norm constraints enforce an absolute upper bound on the magnitude of the incoming weight vector for every neuron and use projected gradient descent to enforce the constraint.
- One of the nice properties is that the parameter vector cannot grow out of control (even if the learning rates are too high) because the updates to the weights are always bounded.

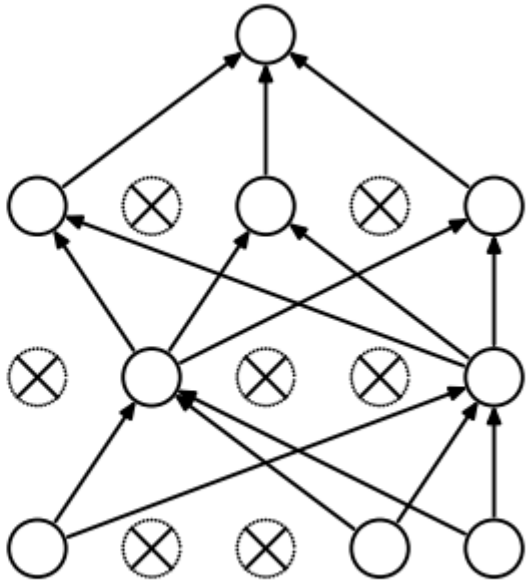
Preventing Overfitting in Deep Neural Networks

- ***Dropout*** is a technique where randomly selected neurons are ignored during training. They are "dropped-out" randomly.
- This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass.
- Thus, we are actually training multiple networks with different compositions of neurons and averaging their results.
- Co-adaptions occur when neurons learn to fix the mistakes made by other neurons on the training data.

Preventing Overfitting in Deep Neural Networks



(a) Standard Neural Net



(b) After applying dropout.

Preventing Overfitting in Deep Neural Networks

- The network thus becomes very good at fitting the training data.
- This prevents network from generalizing to the test data and it becomes more volatile because co-adaptations are tuned to unique characteristics of training data.
- To apply dropout, we need to set a retention probability for each layer. The retention probability specifies the probability that a unit is not dropped.
- For example, if you set the retention probability to 0.8, the units in that layer have an 80% chance of remaining active and a 20% chance of being dropped.

Preventing Overfitting in Deep Neural Networks

- Standard practice is to set the retention probability to 0.5 for hidden layers and to something close to 1, like 0.8 or 0.9 on the input layer. Output layers generally do not apply dropout.
- Dropout is only used during training to make the network more robust to fluctuations in the training data.
- But when it comes to testing, we want to make use of the entire network. In other words, we don't apply dropout to the test data.

Preventing Overfitting in Deep Neural Networks

- This means neurons will receive more connections and therefore more activations during inference than what they were used to during training.
- For example, if we use a dropout rate of 50% dropping two out of four neurons in a layer during training, the neurons in the next layer will receive twice the activations during inference and thus become overexcited.
- Accordingly, the values produced by these neurons will, on average, be too large by 50%.

Preventing Overfitting in Deep Neural Networks

- To correct this overactivation at test and inference time, we multiply the weights of the overexcited neurons by the retention probability (1 – dropout rate) and thus scale them down.
- This implementation of dropout is undesirable because it requires scaling of neuron outputs at test time. Test-time performance is extremely critical to model evaluation.
- An alternative to scaling the activations at test and inference time by the retention probability is to scale them at training time.

Preventing Overfitting in Deep Neural Networks

- We do this by dropping out the neurons and immediately afterward scaling them by the inverse retention probability. This strategy is called inverted dropout.

$$activation \times \frac{1}{retention\ probability}$$

- This operation scales the activations of the remaining neurons up to make up for the signal from the other neurons that were dropped.
- This corrects the activations right at training time and it is often the preferred option.

Preventing Overfitting in Deep Neural Networks

- Dropout regularization technique shines especially when your network is very big or when you train for a very long time, both of which put a network at a higher risk of overfitting.