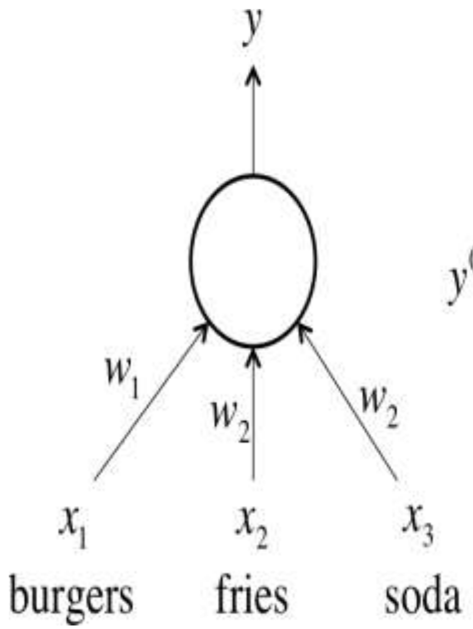
TRAINING FEED FORWARD NEURAL NETWORKS

DR. SANJAY CHATTERJI

The Fast-Food Problem

- During training, we show the neural net a large number of training examples and iteratively modify the weights to minimize the errors we make on the training examples.
- One idea is to be intelligent about picking our training cases.
- Instead, we try to motivate a solution that works well in general.

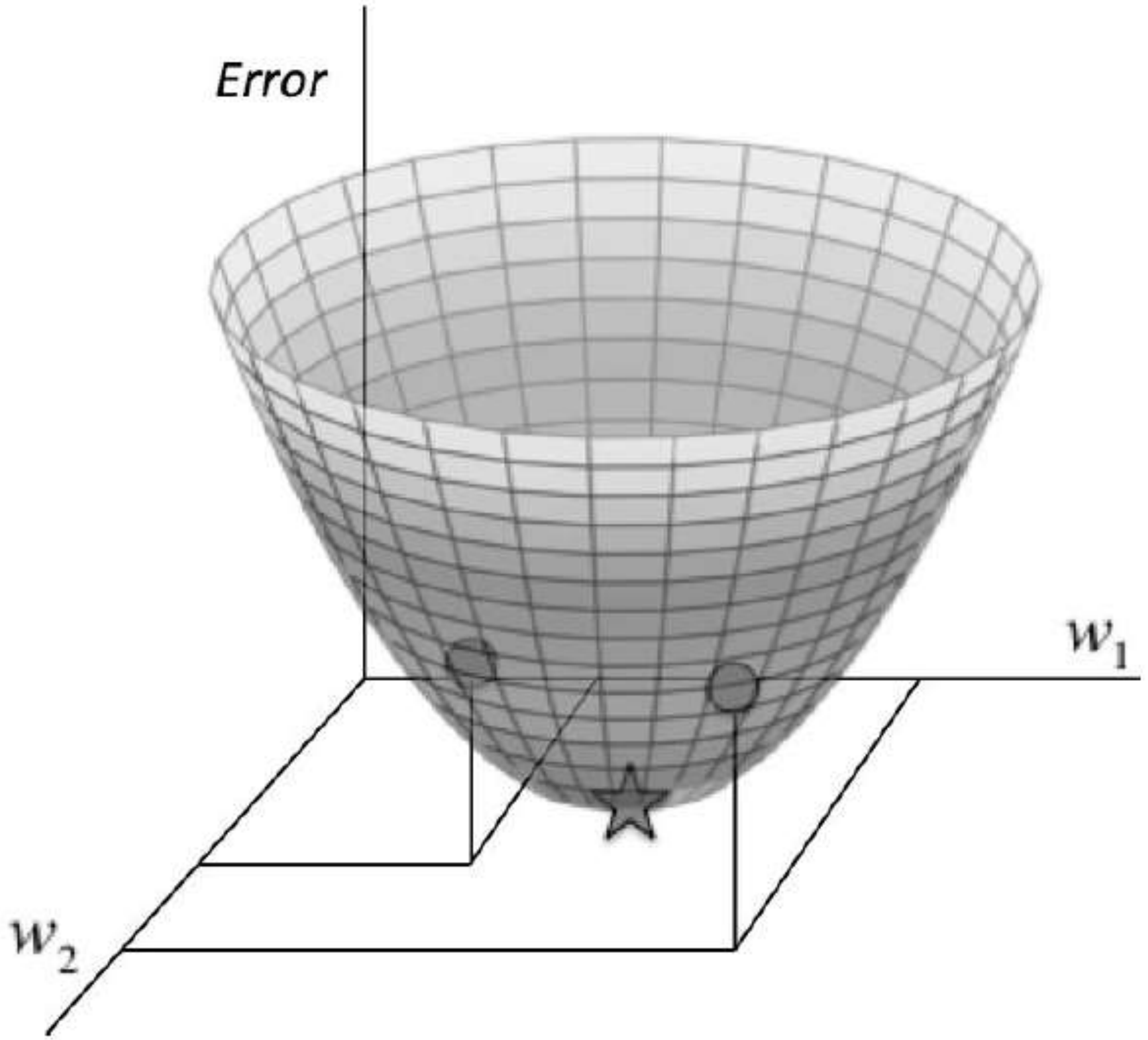


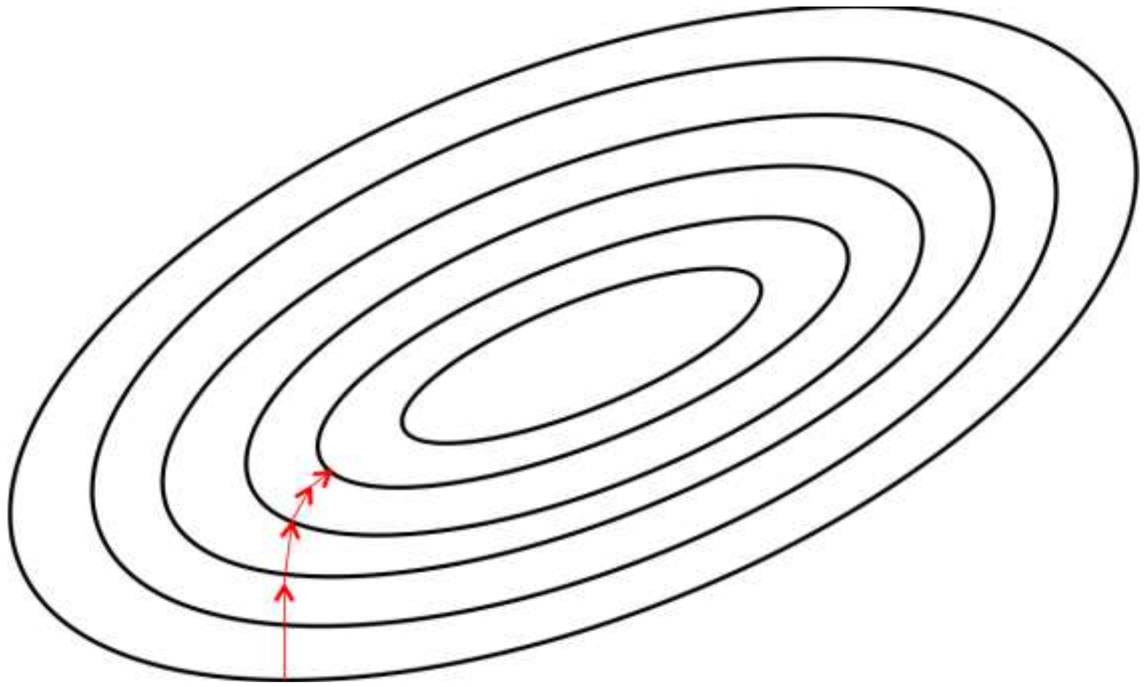
$$y^{(i)} = w_1 x_1^{(i)} + w_2 x_2^{(i)} + w_3 x_3^{(i)}$$

$$E = \frac{1}{2} \Sigma_i \left(t^{(i)} - y^{(i)}\right)^2$$

Gradient Descent

- Minimizing the squared error over all of the training examples by simplifying the problem.
- Linear neuron with two inputs has 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*.
- We can also conveniently visualize this surface as a set of elliptical contours.
- Contours correspond to settings of *w*1 and *w*2 that evaluate to the same value of *E*.
- We can develop a high-level strategy for how to find the values of the weights that minimizes the error function.


Error
w_1
w_2



The Delta Rule and Learning Rates

- Hyperparameters: In addition to the weight parameters defined in our neural network, learning algorithms also require a couple of additional parameters to carry out the training process.
  - *learning rate:* at each step of moving perpendicular to the contour, we need to determine how far we want to walk before recalculating our new direction. It depends on the steepness of the surface.
  - 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.

Continued..

- Now, we are finally ready to derive the *delta rule* for training our linear neuron.
- 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.

$$
\begin{aligned}
\Delta w_k &= -\epsilon \frac{\partial E}{\partial w_k} \\
&= -\epsilon \frac{\partial}{\partial w_k}\left(\frac{1}{2}\Sigma_i \left(t^{(i)} - y^{(i)}\right)^2\right) \\
&= \Sigma_i \epsilon \left(t^{(i)} - y^{(i)}\right)\frac{\partial y_i}{\partial w_k} \\
&= \Sigma_i \epsilon x_k^{(i)}\left(t^{(i)} - y^{(i)}\right)
\end{aligned}
$$

Gradient Descent with Sigmoidal Neurons

- Now, we will deal with training neurons and neural networks that utilize nonlinearities.
- We use the sigmoidal neuron as a model.
- For simplicity, we assume that the neurons do not use a bias term.
- The neuron computes the weighted sum of its inputs, the logit *z*.
- It then feeds its logit into the input function to compute *y*, its final output.

$$z = \sum_k w_k x_k$$

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

Gradient Descent with Sigmoidal Neurons

- For learning, we want to compute the gradient of the error function with respect to the weights.
- To do so, we start by taking the derivative of the logit with respect to the inputs and the weights:
- The update rule is just like the delta rule, except with extra multiplicative terms.

$$\frac{\partial z}{\partial w_k} = x_k$$

$$\frac{\partial z}{\partial x_k} = w_k$$

$$\begin{aligned}\frac{dy}{dz} &= \frac{e^{-z}}{\left(1+e^{-z}\right)^2} \\ &= \frac{1}{1+e^{-z}}\frac{e^{-z}}{1+e^{-z}} \\ &= \frac{1}{1+e^{-z}}\left(1-\frac{1}{1+e^{-z}}\right) \\ &= y(1-y)\end{aligned}$$

$$\frac{\partial y}{\partial w_k} = \frac{dy}{dz}\frac{\partial z}{\partial w_k} = x_k y(1-y)$$

$$\frac{\partial E}{\partial w_k} = \sum_i \frac{\partial E}{\partial y^{(i)}}\frac{\partial y^{(i)}}{\partial w_k} = -\sum_i x_k^{(i)} y^{(i)}\left(1-y^{(i)}\right)\left(t^{(i)}-y^{(i)}\right)$$

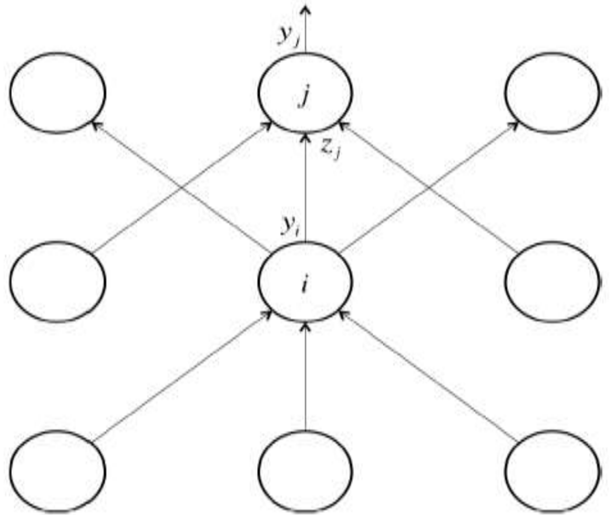
$$\Delta w_k = \sum_i \epsilon x_k^{(i)} y^{(i)}\left(1-y^{(i)}\right)\left(t^{(i)}-y^{(i)}\right)$$

The Backpropagation Algorithm

- How to tackle the problem of training multilayer neural networks (instead of just single neurons)?
- We don’t know what the hidden units ought to be doing.
- We can compute how fast the error changes as we change a hidden activity.
- From there, we can figure out how fast the error changes when we change the weight of an individual connection.
- Essentially, we’ll be trying to find the path of steepest descent in high dimensional space!

The Backpropagation Algorithm Cont.

- Our strategy will be one of dynamic programming.
- From ED(one layer of hidden units) we compute ED(activities of the below layer) and then we compute ED(weights leading into the unit). ED: Error Derivative
- Suppose we have the error derivatives for layer j.
- We now aim to calculate the error derivatives for the layer below it, layer i.
- To do so, we must accumulate information about how th output of a neuron in layer i affects the logits of every neuron in layer j. This can be done as follows



$$E = \frac{1}{2}\sum_{j \in output}\left(t_j - y_j\right)^2 \Rightarrow \frac{\partial E}{\partial y_j} = -\left(t_j - y_j\right)$$

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial E}{\partial z_j}\frac{dz_j}{dy_i} = \sum_j w_{ij}\frac{\partial E}{\partial z_j}$$

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j}\frac{dy_j}{dz_j} = y_j\left(1 - y_j\right)\frac{\partial E}{\partial y_j}$$

$$\frac{\partial E}{\partial y_i} = \sum_j w_{ij} y_j\left(1 - y_j\right)\frac{\partial E}{\partial y_j}$$

The Backpropagation Algorithm Cont.

- We finally express the error derivatives of layer i in terms of the error derivatives of layer j.
- Once we've gone through the whole dynamic programming routine, having filled up the table appropriately with all of our partial derivatives, we can determine how the error changes with respect to the weights.

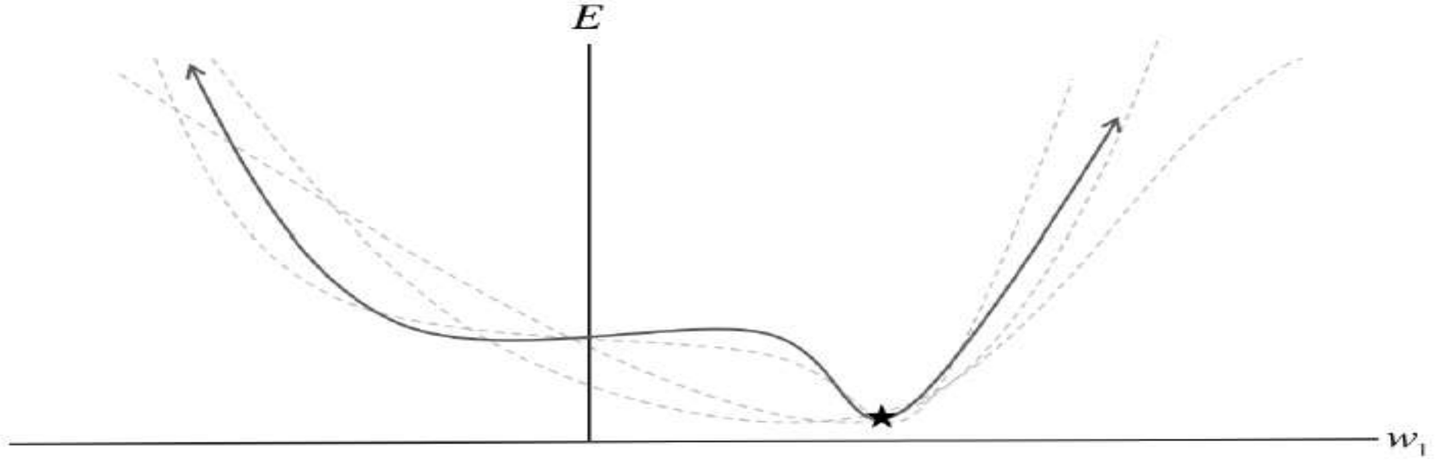
$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}} \frac{\partial E}{\partial z_j} = y_i y_j \left(1 - y_j\right) \frac{\partial E}{\partial y_j}$$

- This gives us how to modify the weights after each training example:
- Finally, to complete the algorithm, just as before, we merely sum up the partial derivatives over all the training examples in our dataset. This gives us the following modification formula:

$$\Delta w_{ij} = -\sum_{k \in dataset} \epsilon y_i^{(k)} y_j^{(k)} \left(1 - y_j^{(k)}\right) \frac{\partial E^{(k)}}{\partial y_j^{(k)}}$$

Stochastic and Minibatch Gradient Descent

- Here we've been using a version of gradient descent known as *batch gradient descent*.
- Another potential approach is *stochastic gradient descent*, where at each iteration error surface is estimated only with respect to a single example.
- Instead of a single static error surface, here our error surface is dynamic.
- In mini-batch gradient descent, at every iteration, we compute the error surface with respect to some subset of the total dataset.

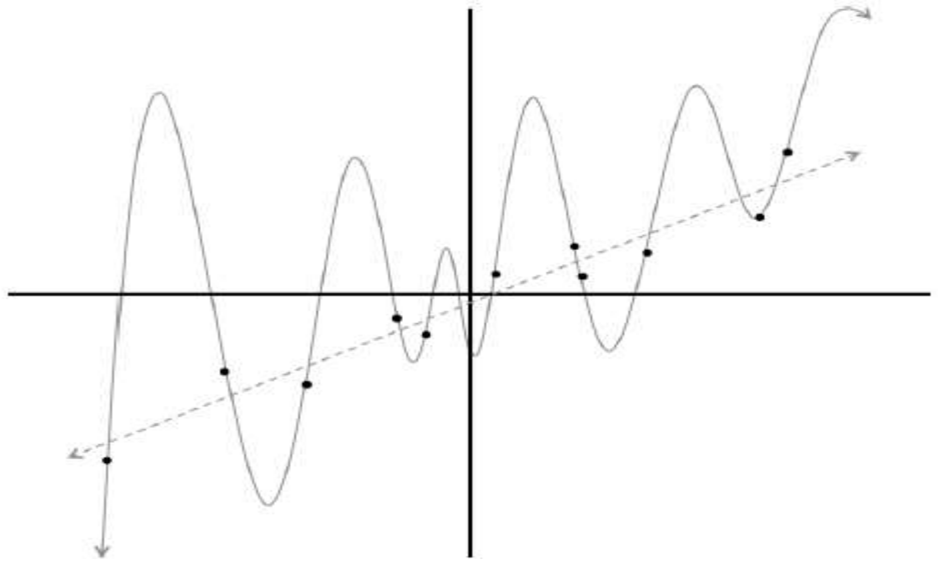


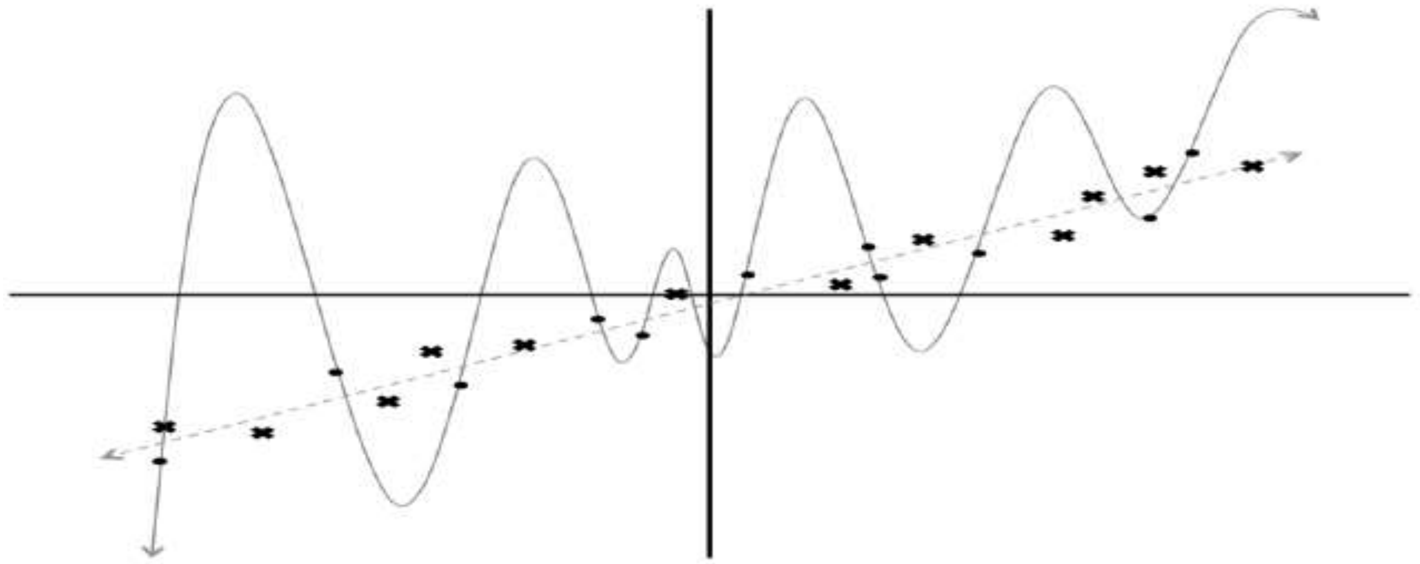
Test Sets, Validation Sets, and Overfitting

- One of the major issues with artificial neural networks is that the models are quite complicated.
- 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.
- We are given a bunch of data points on a flat plane, and our goal is to find a curve that best describes this dataset.
- Using the data, we train two different models: a linear model and a degree 12 polynomial.

Which curve should we trust?

- Which curve should we trust?
  - The line which gets almost no training example correctly?
  - Or the complicated curve that hits every single point in the dataset?
- At this point we might trust the linear fit because it seems much less contrived.
- But just to be sure, let’s add more data to our dataset!



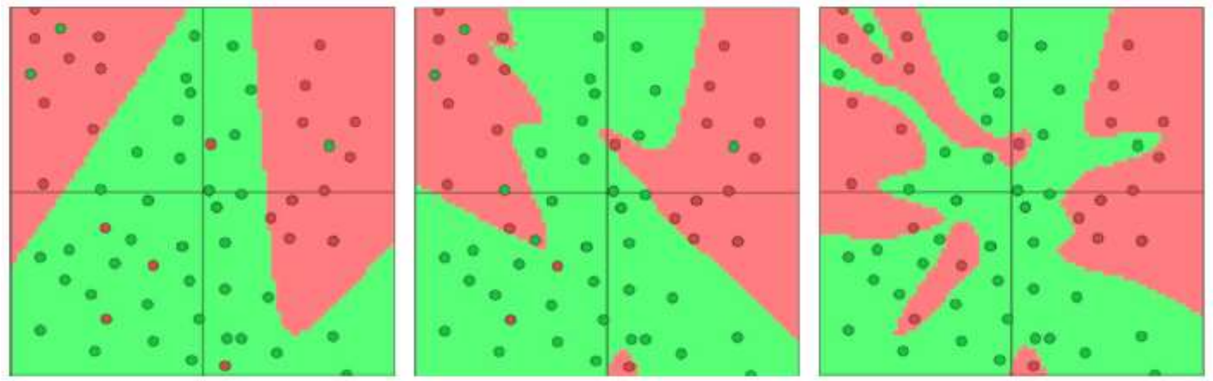


Overfitting

- Linear model is better subjectively and also quantitatively (measured using the squared error metric).
- By building a very complex model, it’s quite easy to perfectly fit our training dataset. But when we evaluate such a complex model on new data, it performs very poorly.
- The complex model does not *generalize* well => Overfitting.
- It is one of the biggest challenges that a machine learning engineer must combat.
- This becomes 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.

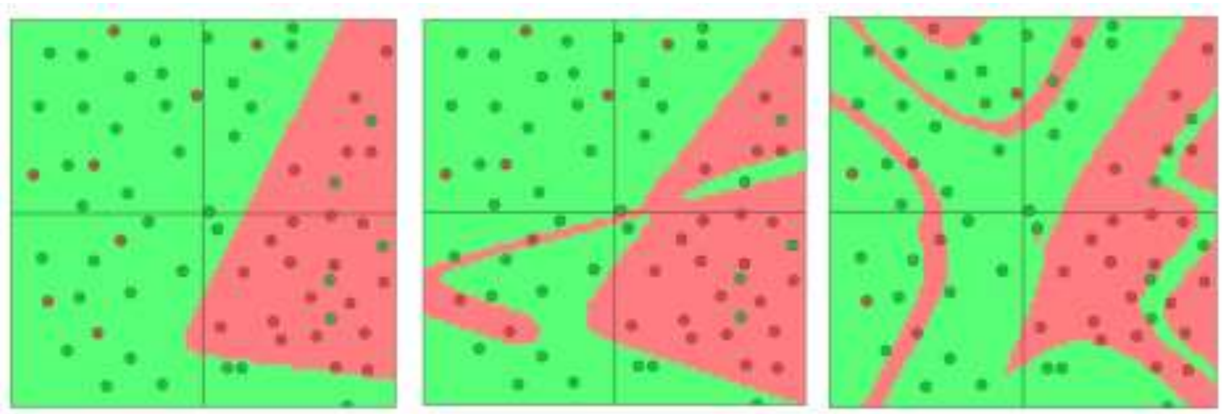
Let’s see how this looks in the context of a neural network

- ANN with two inputs, a softmax output of size two, and a hidden layer with 3, 6, or 20 neurons.
- We train these networks using mini-batch gradient descent (batch size 10), and the results visualized using ConvNetJS.
- It’s apparent from the images that as the number of connections in our network increases, so does our propensity to overfit to the data.



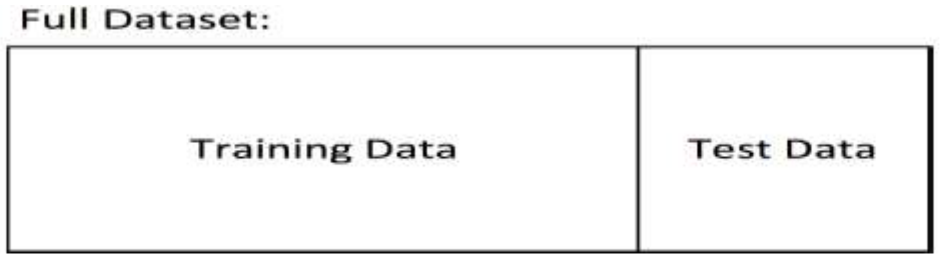
As no. of connections and depth increases, propensity to overfit also increases

- We can similarly see the phenomenon of overfitting as we make our neural networks deep.
- ANN that have one, two, or four hidden layers of three neurons each.



Three major observations

1. 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, we run the risk of overfitting. Deep learning takes the approach of solving very complex problems with complex models and taking additional countermeasures to prevent overfitting.
2. It is very misleading to evaluate a model using the data we used to train it. Using some examples, we would falsely suggest that the degree 12 polynomial model is preferable to a linear fit. 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.



Three major observations

Point 2 Contd..

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. In the real world, large datasets are hard to come by, so it might seem like a waste to not use all of the data at our disposal during the training process. Consequently, it may be very tempting to reuse training data for testing or cut corners while compiling test data. Be forewarned: if the test set isn't well constructed, we won't be able draw any meaningful conclusions about our model.

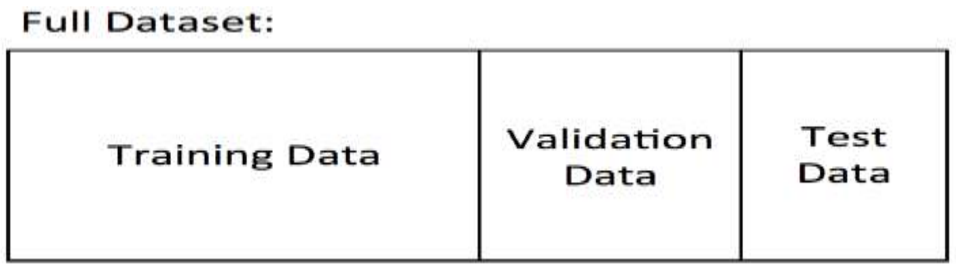
3. It's quite likely that while we're training our data, there's a point in time where instead of learning useful features, we start **overfitting** to the training set.

Avoiding Overfitting

- **To avoid that**, we want to be able to stop the training process as soon as we start overfitting, to prevent poor generalization.
- To do this, we divide our training process into **epochs**. An epoch is a single iteration over the entire training set. In other words, if we have a training set of size d and we are doing mini-batch gradient descent with batch size b, then an epoch would be equivalent to d/b model updates.
- 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**. 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.

Hyperparameter Optimization

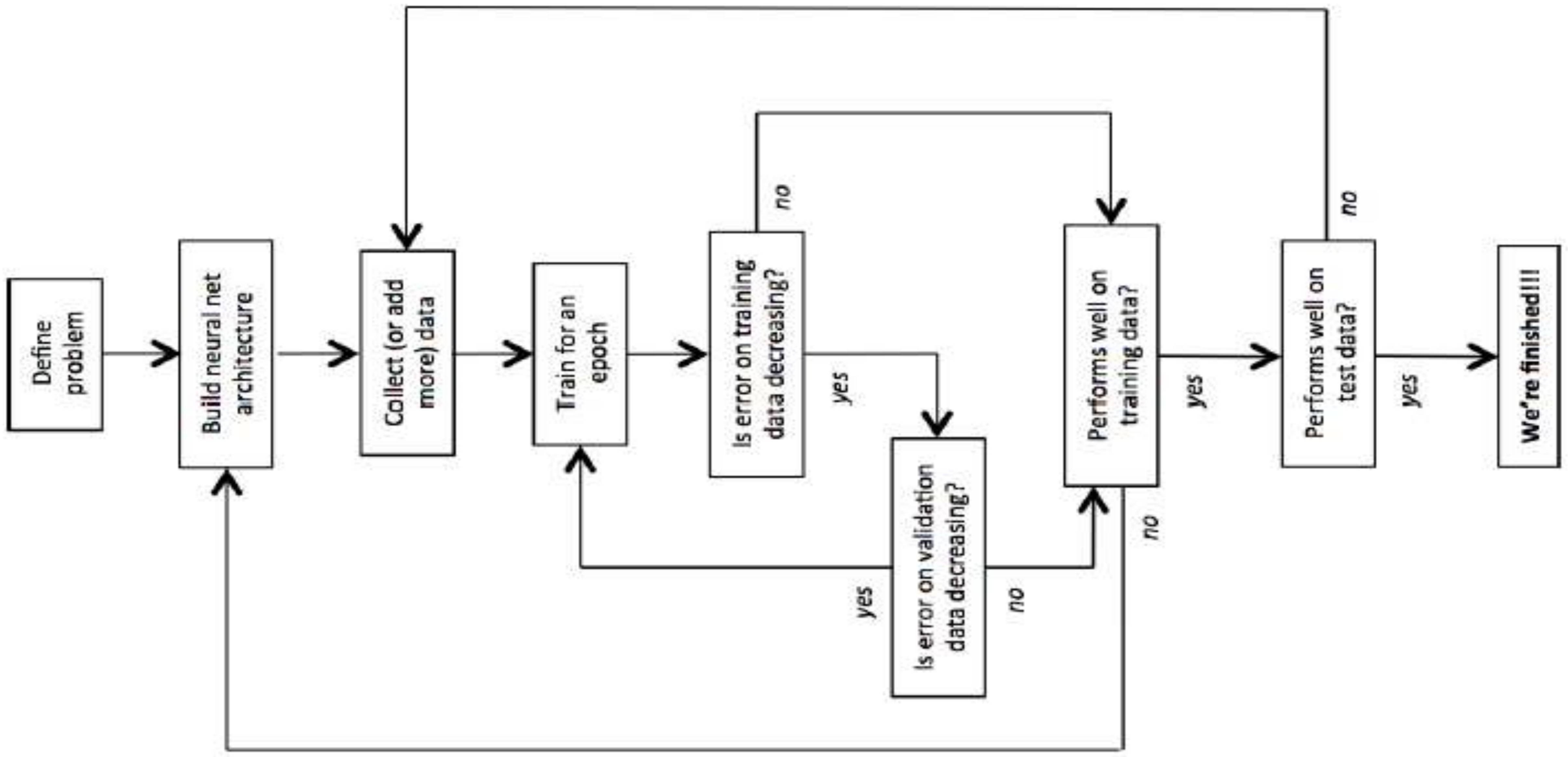
- The validation set is also helpful as a proxy measure of accuracy during the process of hyperparameter optimization.
- One potential way to find the optimal setting of hyperparameters is by applying a grid search, where we pick a value for each hyperparameter from a finite set of options (e.g., LR∈0.001, 0.01, 0.1, batch size∈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.



Lets define our problem rigorously

- Let’s say our goal was to train a deep learning model to identify cancer.
- Our input would be an RBG image, which can be represented as a vector of pixel values.
- Our output would be a probability distribution over three mutually exclusive possibilities: 1) normal, 2) benign tumor or 3) malignant tumor.
- After we define our problem, we need to build a neural network architecture to solve it.
- Our input layer would have to be of appropriate size to accept the raw data from the image, and our output layer would have to be a softmax of size 3.
- We also want to collect a significant amount of data for training or modeling.
- This data would probably be in the form of uniformly sized pathological images that have been labeled by a medical expert.
- We shuffle and divide this data into separate training, validation, and test sets.

workflow we use when building and training deep learning models


Define problem
Build neural net architecture
Collect (or add more) data
Train for an epoch
Is error on training data decreasing?
no
yes
Is error on validation data decreasing?
yes
no
Performs well on training data?
yes
no
Performs well on test data?
no
yes
We're finished!!!

Begin gradient descent

- We train the model on our training set for an epoch at a time.
- At the end of each epoch, we ensure that our error on the training set and validation set is decreasing.
- When one of these stops to improve, we terminate and make sure we’re happy with the model’s performance on the test data.
- If we’re unsatisfied, we need to rethink our architecture or reconsider whether the data we collect has the information required to make the prediction we’re interested in making.
- If our training set error stopped improving, we probably need to do a better job of capturing the important features in our data.

Gradient descent

- If our validation set error stopped improving, we probably need to take measures to prevent overfitting.
- If, however, we are happy with the performance of our model on the training data, then we can measure its performance on the test data, which the model has never seen before this point.
- If it is unsatisfactory, we need more data in our dataset because the test set seems to consist of example types that weren’t well represented in the training set.
- Otherwise, we are finished!

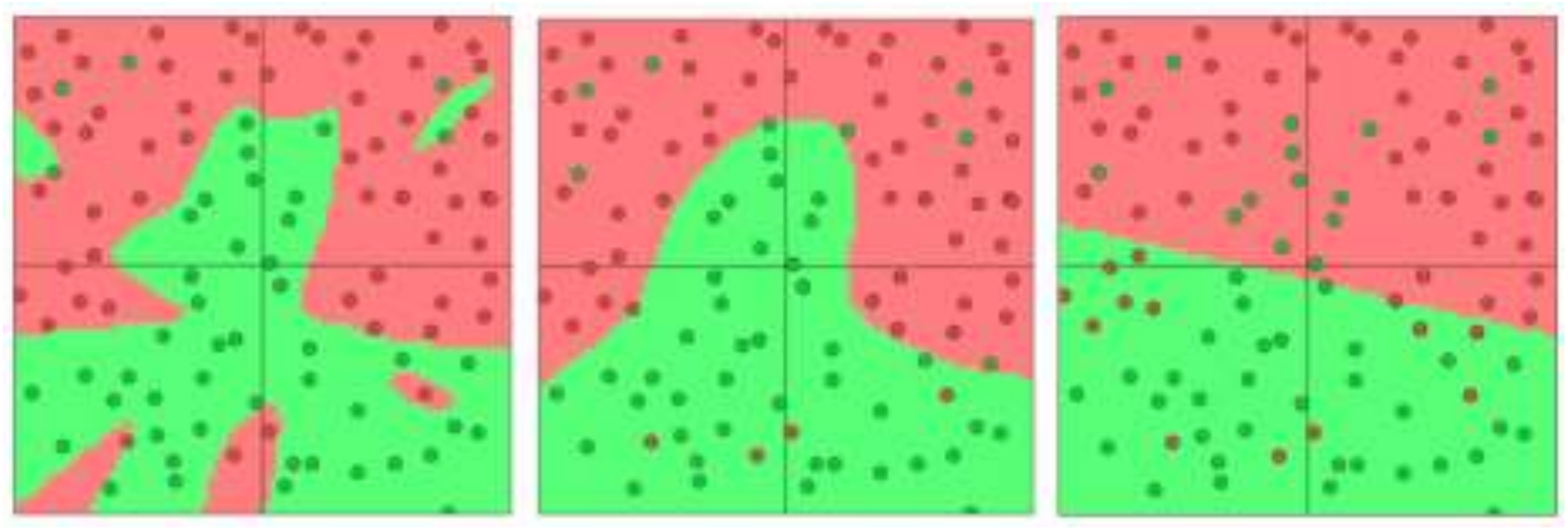
Preventing Overfitting in Deep Neural Networks

- One method of combatting overfitting is called *regularization*.
- Regularization modifies the objective function that we minimize by adding additional terms that penalize large weights.
- We change the objective function so that it becomes *Error + λ f(θ)*
  - *f(θ) grows larger as the components of θ grow larger.*
  - *λ* is the regularization strength
- The value we choose for λ determines how much we want to protect against overfitting.
  - *λ = 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.*

L2 regularization

- *The most common type of regularization in machine learning is L2 regularization.*
- *It can be implemented by augmenting the error function with the squared magnitude of all weights in the neural network.*
- *L2 regularization:* we add $\frac{1}{2} \lambda w^2$ to the error function.
- The L2 regularization has the intuitive interpretation of heavily penalizing peaky weight vectors and preferring diffuse weight vectors.
- 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.
- Of particular note is that during the gradient descent update, using the L2 regularization ultimately means that every weight is decayed linearly to zero.
- Because of this phenomenon, L2 regularization is also commonly referred to as weight decay.

ANN using mini-batch gradient descent (batch size 10) and L2 regularization strengths of 0.01, 0.1, and 1



L1 regularization

- *Another common type of regularization is L1 regularization.*
- *L1 regularization:* we add the term $\lambda|w|$.
- The value of λ determines how much we want to protect overfitting.
  - $\lambda = 0$ ➔ we do not take any measures against the possibility of overfitting.
  - λ is too large ➔ our model will prioritize keeping θ as small as possible over trying to find the parameter values that perform well on our training set.

L1 Vs L2 Regularization

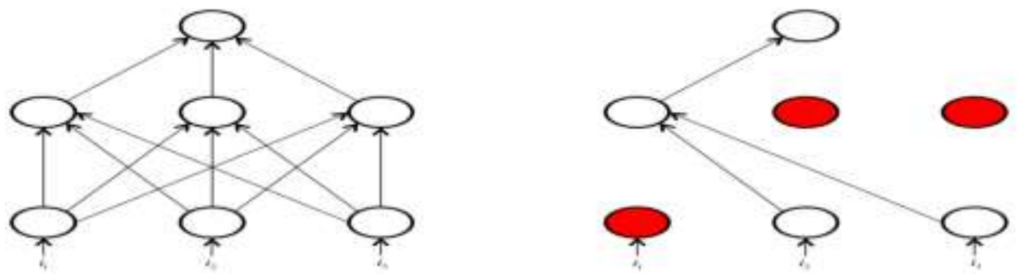
- The L1 regularization leads the weight vectors to become sparse during optimization (close to zero). So, neurons with L1 regularization end up using only a small subset of their most important inputs and become quite resistant to noise.
- In comparison, weight vectors from L2 regularization are usually diffuse, small numbers.
- L1 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 L2 regularization because it empirically performs better.

Max norm constraints

- They have the goal of attempting to restrict θ from becoming too large.
- They do this more directly.
- They enforce an absolute upper bound c on the magnitude of the incoming weight vector for every neuron and use projected gradient descent to enforce the constraint.
- Typical values of *c* are 3 and 4.
- One of the nice properties is that the parameter vector cannot grow out of control (even if learning rates are too high) because the updates to the weights are bounded.

Dropout

- Dropout is implemented by only keeping a neuron active with some probability *p.*
- It prevents the network from becoming too dependent on any one (or any small combination) of neurons.
- Mathematically, it prevents overfitting by providing a way of approximately combining exponentially many different neural network architectures efficiently.
- Inverted dropout: neuron whose activation hasn't been silenced has its output divided by *p* before the value is propagated to the next layer.



Intricacies

- Dropout is pretty intuitive to understand, but there are some important intricacies to consider.
- First, we'd like the outputs of neurons during test time to be equivalent to their expected outputs at training time.
- We could fix this naively by scaling the output at test time.
- For example, if $p = 0.5$, neurons must halve their outputs at test time in order to have the same (expected) output they would have during training.
- This means that if a neuron's output prior to dropout was x, then after dropout, the expected output would be E output $= px + (1-p) \cdot 0 = px$.

Intricacies

- This naive implementation of dropout is undesirable, however, because it requires scaling of neuron outputs at test time.
- Test-time performance is extremely critical to model evaluation, so it’s always preferable to use *inverted dropout*, where the scaling occurs at training time instead of at test time.
- In inverted dropout, any neuron whose activation hasn’t been silenced has its output divided by p before the value is propagated to the next layer.
- With this fix, $E\,[\text{output}] = p \cdot x/p + (1 - p) \cdot 0 = x$, and we can avoid arbitrarily scaling neuronal output at test time.

Thank You