UNIVERSITY OF TORONTO
SCHOOL OF CONTINUING STUDIES

3546 – Deep Learning

Module 2: Tuning Neural Networks

Course Syllabus

Module	Topic	Deliverables
1	Course Intro + Review	Term Project Released
2	Model Tuning	Assignment 1 Released
3	Convolutional Networks	
4	Deep Computer Vision	Assignment 1 Due, A2 Released
5	Recurrent Neural Networks	
6	Natural Language Processing	
7	Deep Models for Text	Assignment 2 Due, A3 Released
8	Representational Learning & Variational Methods	
9	Deep Generative Models	Assignment 3 Due, A4 Released
10	Speech and Music Recognition & Synthesis	
11	Term Project Presentations A	Term Project Due
12	Term Project Presentations B	Assignment 4 Due

Learning Outcomes for this Module

Part 1: Module 1 Spillover

- Recall foundational concepts from Machine Learning.
- Understand the anatomy of a neural network.
- Review how neural networks are trained.
- Be able to train a neural network with Keras.

Part 2: Model Tuning

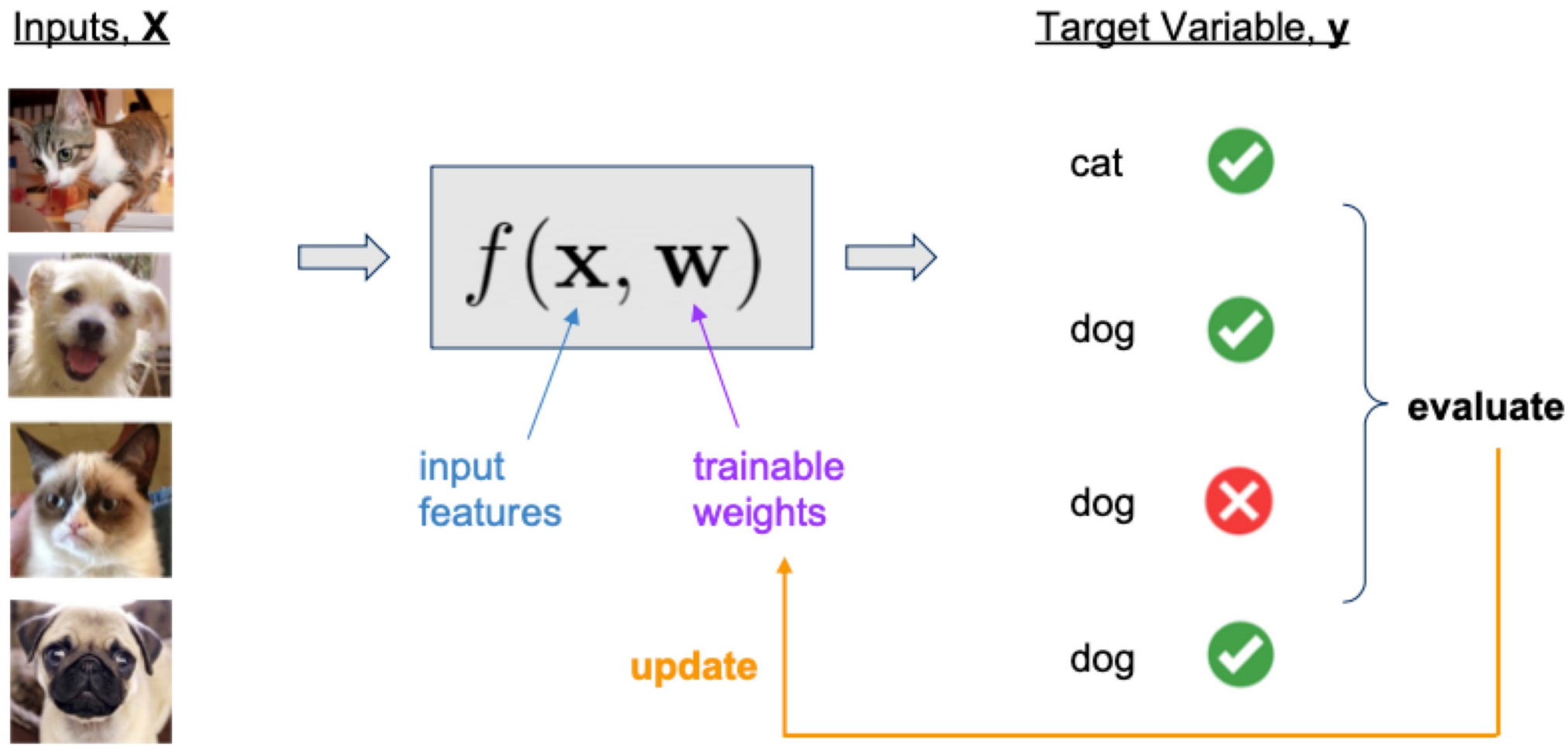
- Understand the vanishing gradient problem and how to mitigate it.
- Explore better optimizers for faster training and escaping local minima.
- Be able to apply regularization techniques to improve generalization performance.
- Learn how to use manual and automated approaches for neural network tuning.

Module 2 – Section 1

Review of Neural Networks

Review: Supervised Learning

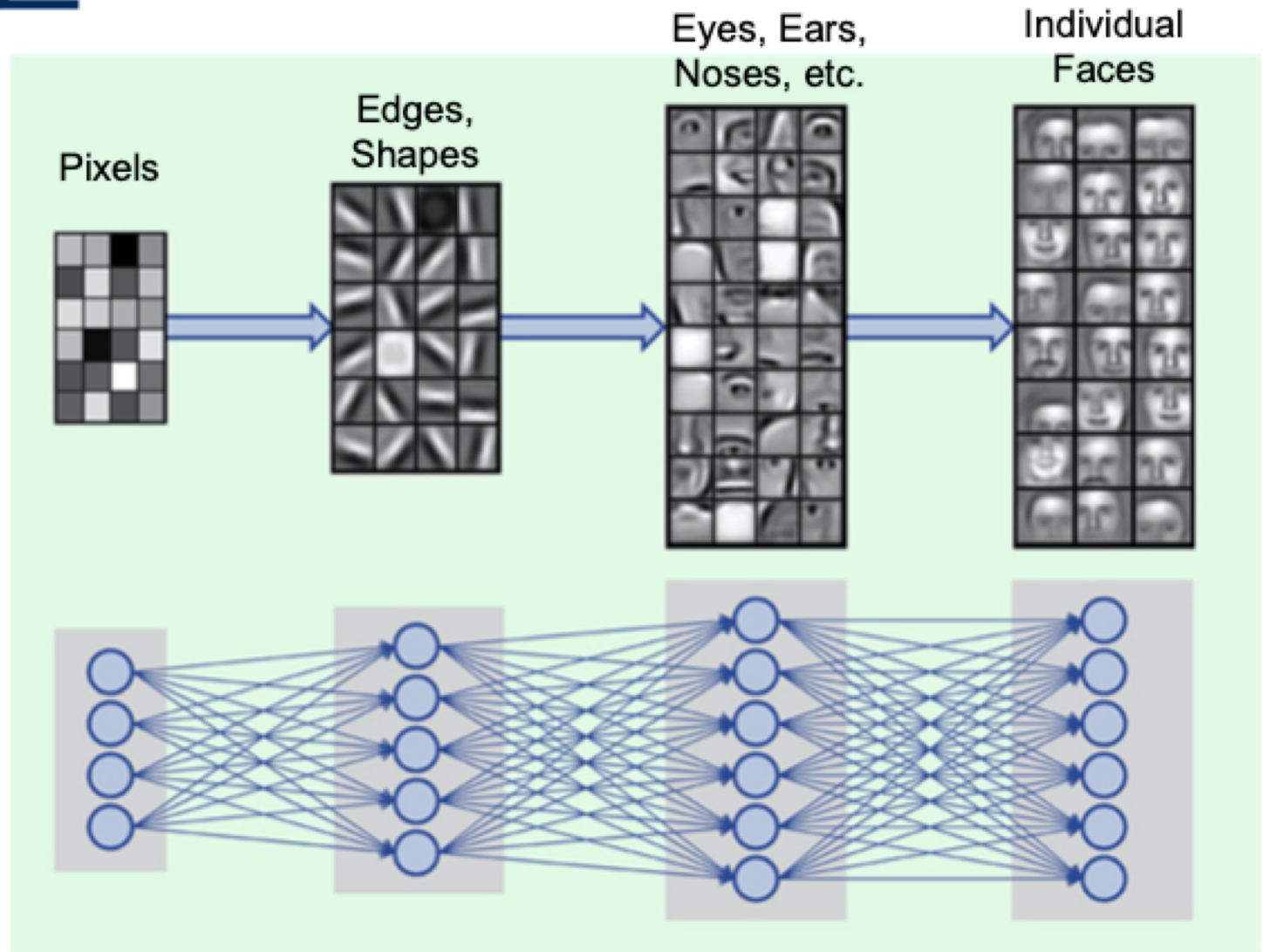
Learn a tunable function f(x, w) that maps input values to known outputs.



Why Deep Learning?

Traditional ML relies on ‘hand-engineered’ features.

Neural networks learn the underlying features directly from the data, in a **hierarchical manner**.



Source: Adopted from Lee et. al, https://doi.org/10.1145/2001269.2001295

Why Now?

Three factors are driving the recent boom in Deep Learning:

1. Bigger Datasets

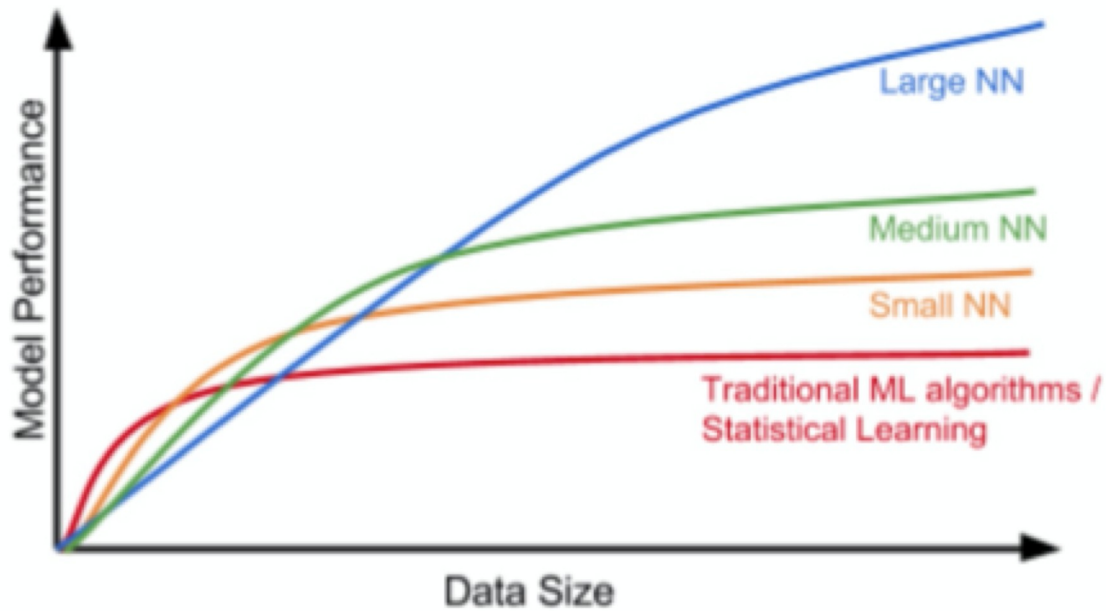


Figure 1. Neural Network Performance vs Other Algos

Source: University of Waterloo

2. Greater parallelization

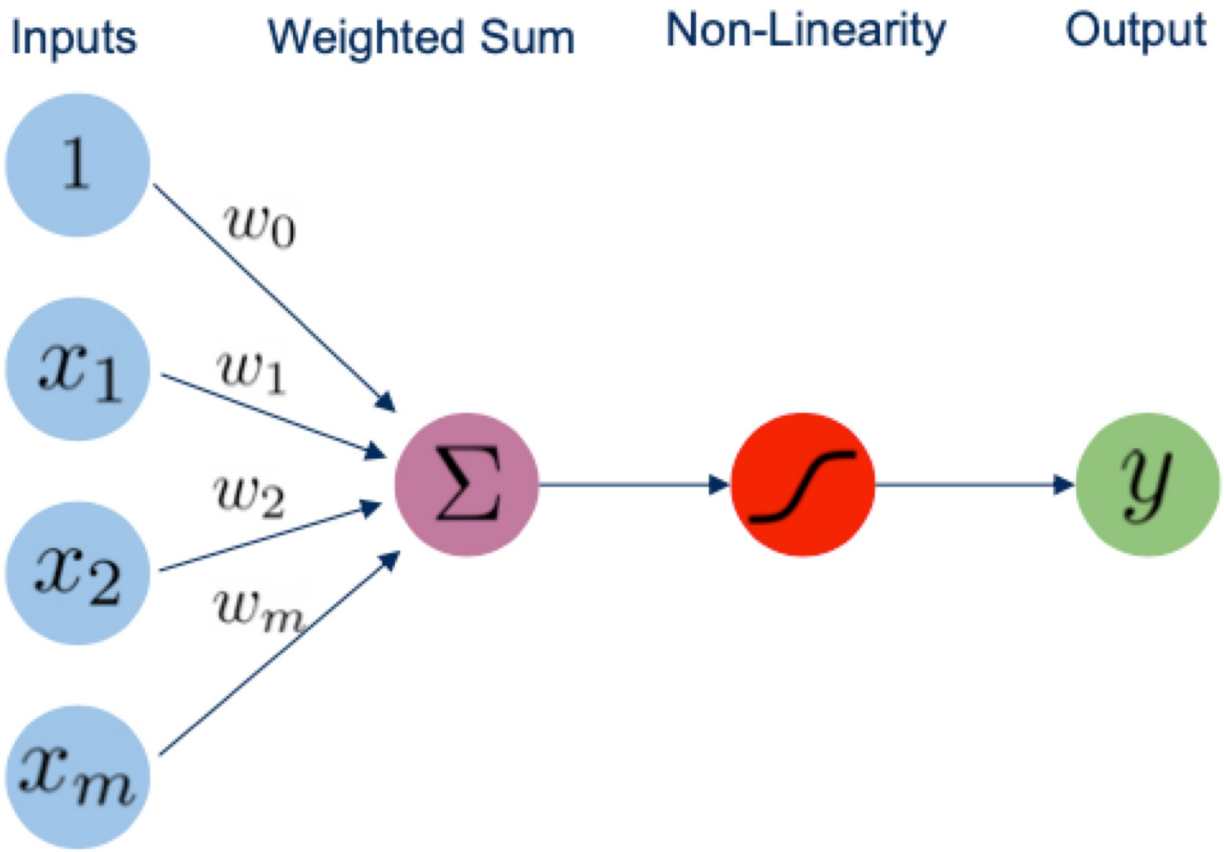
e.g. GPUs

3. Better frameworks

e.g. TensorFlow

PyTorch

A Single Neuron



Nonlinear activation function

Bias

$$y = f\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$$

In vector form:

$$y = f\left(w_0 + \mathbf{X}^\mathbf{T}\mathbf{W}\right)$$

$$\mathbf{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \quad \mathbf{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$$

Activation Functions

Sigmoid:

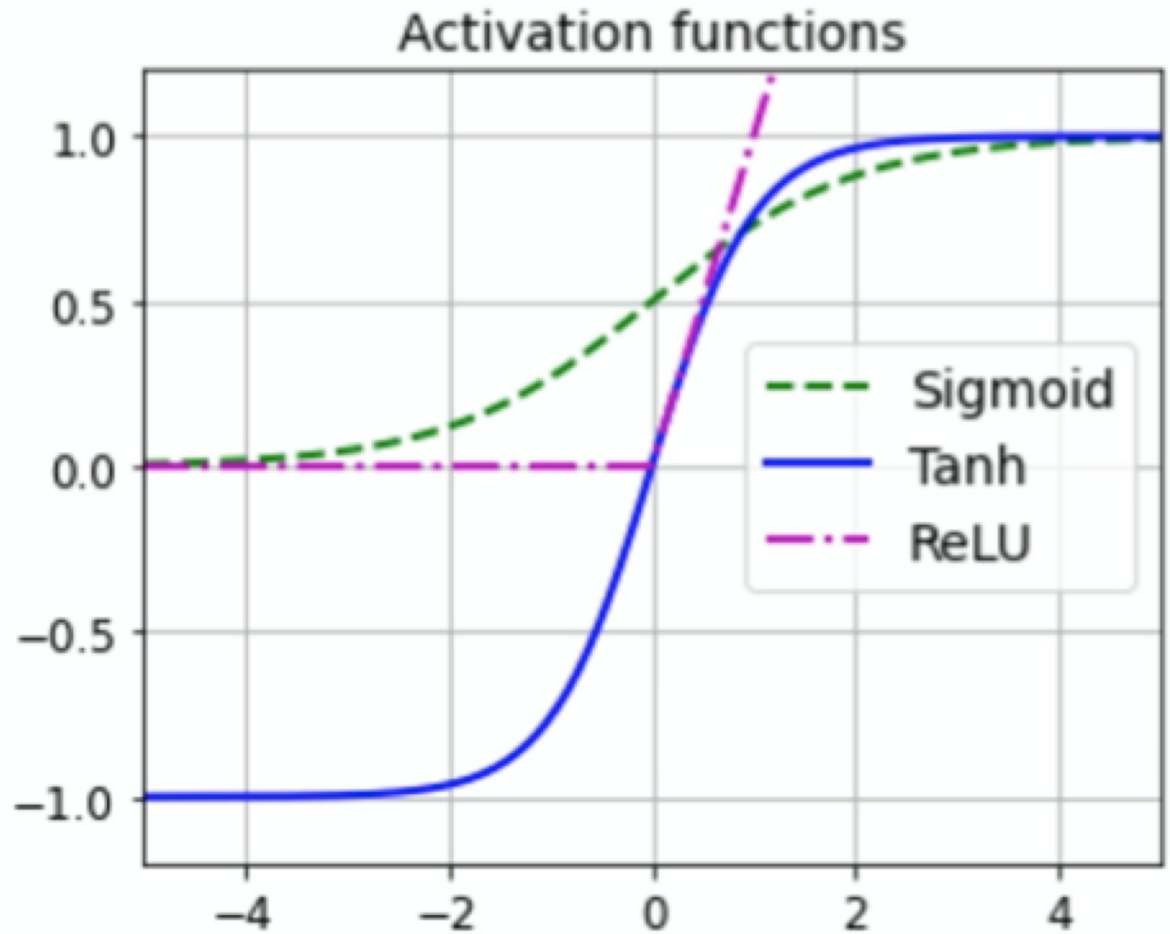
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Hyperbolic Tangent:

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

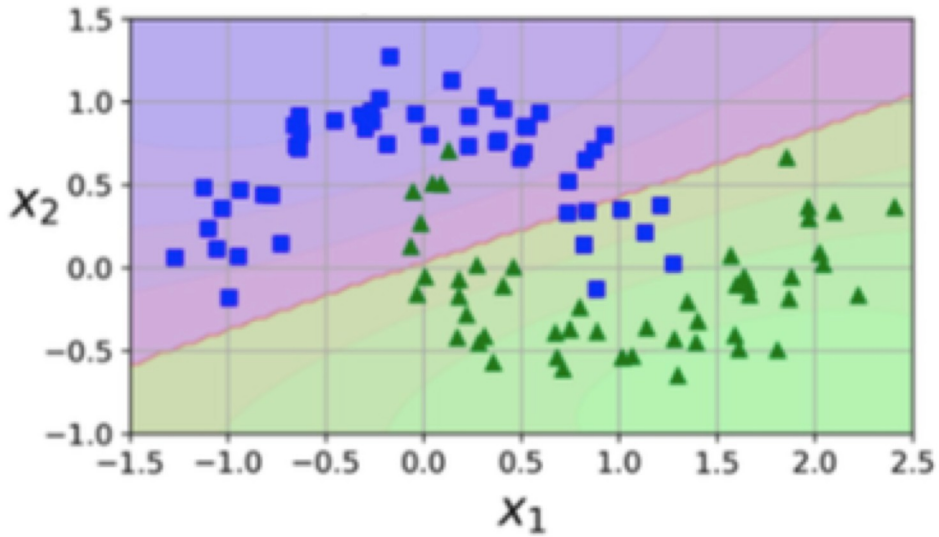
Rectified Linear Unit:

$$ReLU(x) = max(x, 0)$$

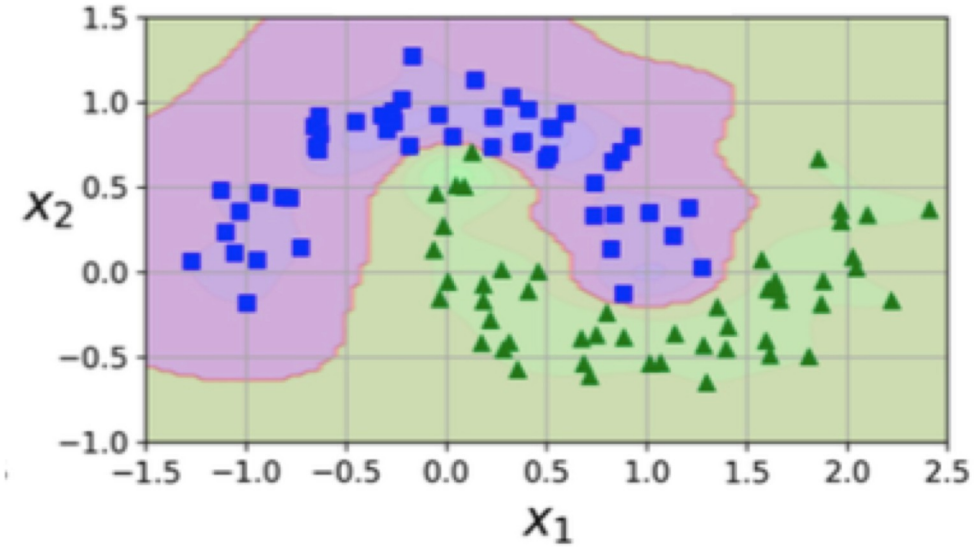


Why Do We Need Activation Functions?

They allow us to approximate arbitrarily complex functions, to yield **non-linear decision boundaries**.



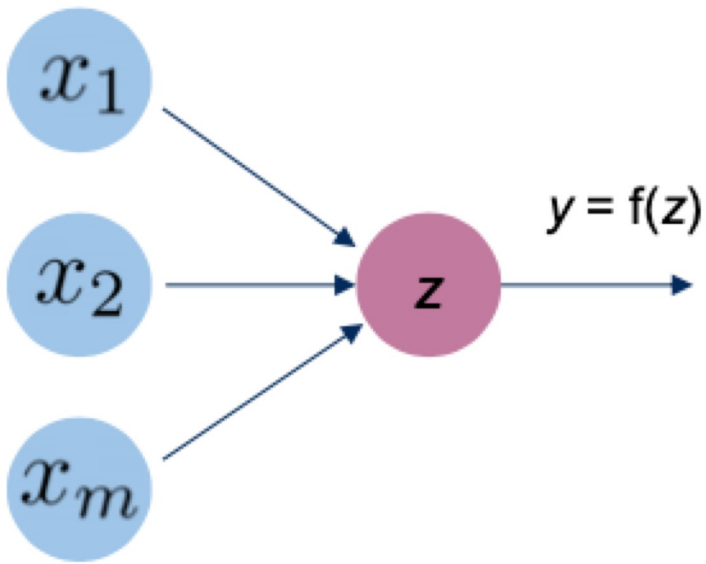
Without nonlinear activations, we get **linear** decision boundaries for any network.



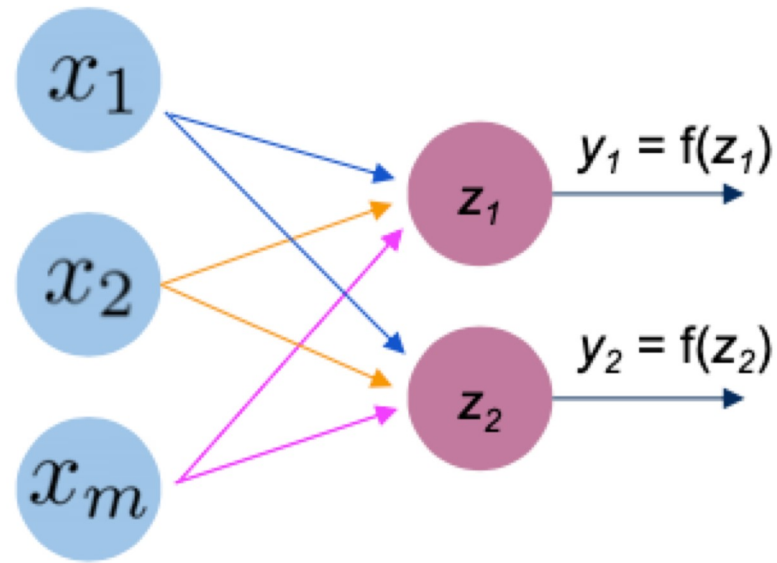
With nonlinear activations, we can approximate any function.

From Neurons to Networks

- Every arrow from x_i to z_j has a corresponding weight w_{ij}
- We don't show the bias, for simplicity, but it is still there
- 'z' is the (dot product + bias) *before* the nonlinearity
- 'y' is the output *after* the nonlinearity (called the 'activation')

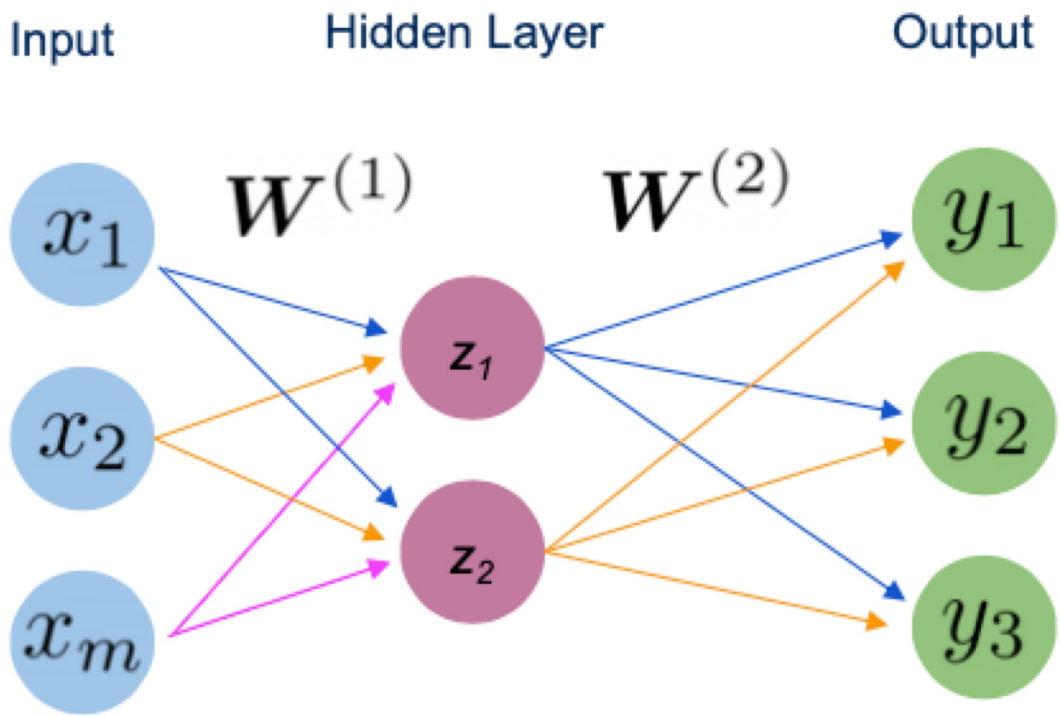


$$y = f\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$$



$$y_j = f\left(w_{0,j} + \sum_{i=1}^{m} x_i w_{i,j}\right)$$

Single-Layer Neural Network



Each layer of weights gets its own weight matrix $\boldsymbol{W}^{(k)}$

The activations of the first layer become the inputs to the next.

$$z_j = w_{0,j}^{(1)} + \sum_{i=1}^{m} x_i w_{i,j}^{(1)}$$

$$y_j = f\left(w_{0,j}^{(2)} + \sum_{i=1}^{m} f(z_i) w_{i,j}^{(2)}\right)$$

Deep Neural Network

Any neural network with two or more layers is considered a deep neural net.

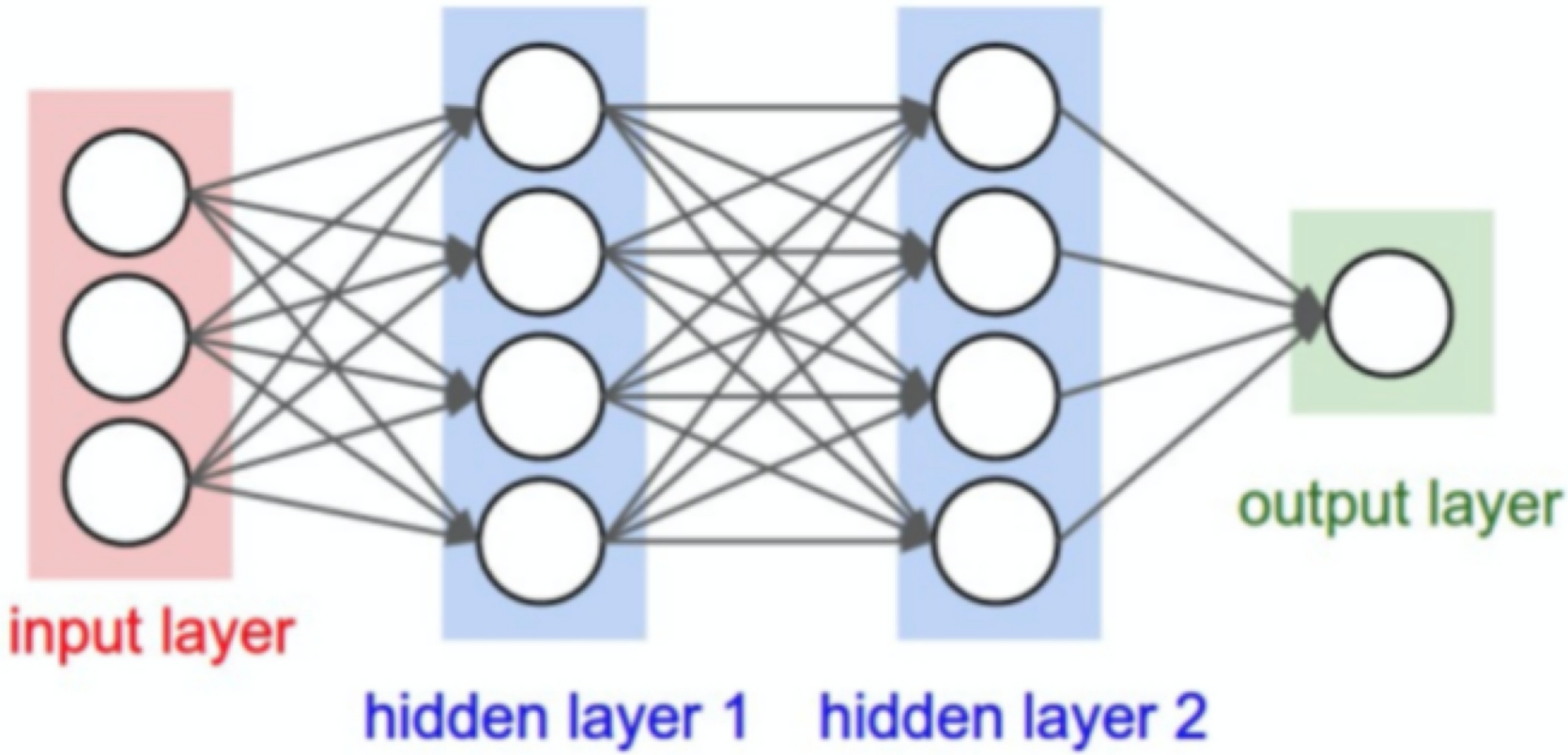


Figure 4. A basic feedforward neural network

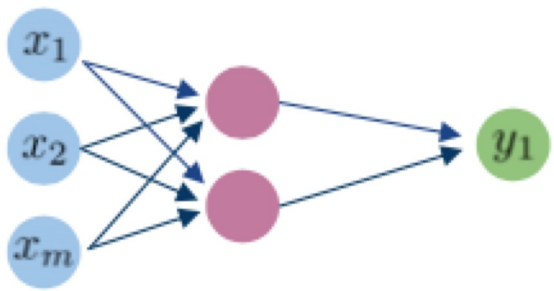
Source: University of Waterloo

Output Layer Activation Functions

We typically use the same activation function (e.g. ReLU) for all hidden layers. However, for our output layer, our choice depends on the task at hand.

Regression

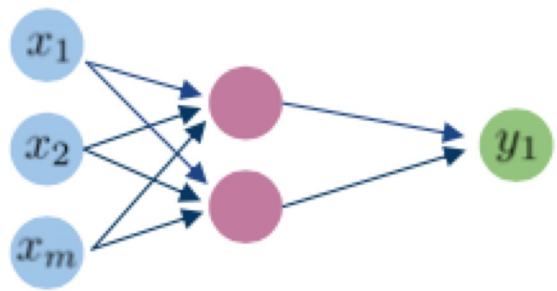
Single output neuron,
No activation



ReLU None

Binary Classification

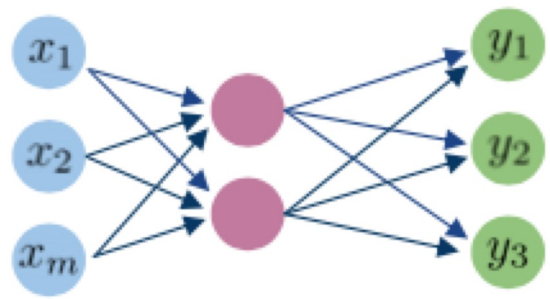
Single output neuron,
Sigmoid activation



ReLU Sigmoid

Multiclass Classification

K output neurons (for K classes),
Softmax activation



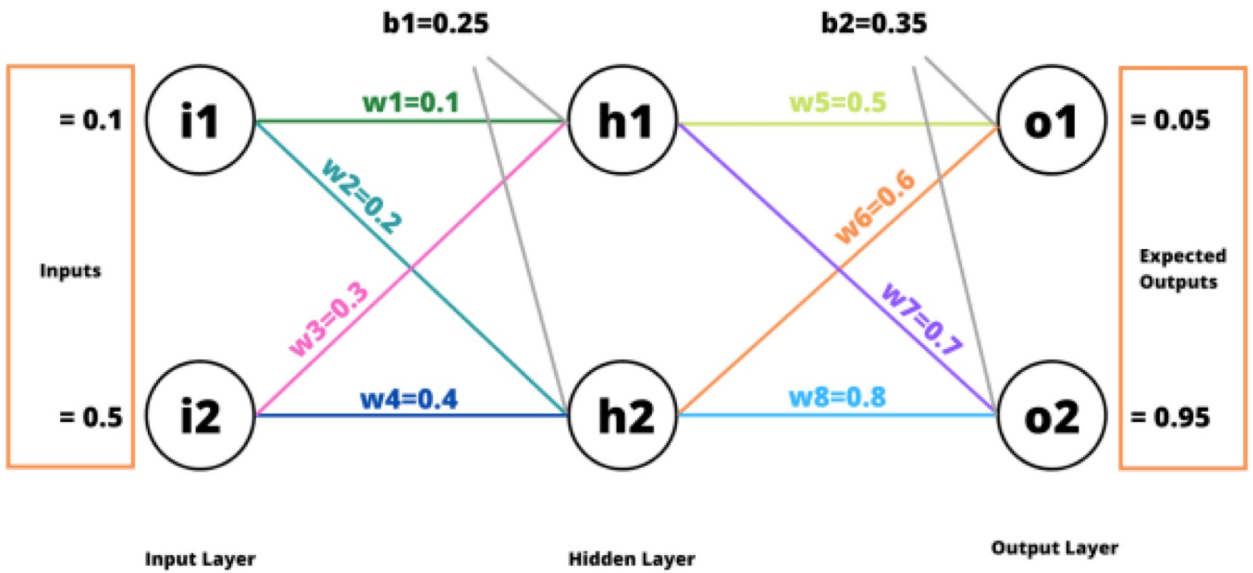
ReLU Softmax

Gradient Descent

- Standard optimization algorithm

- facilitates the search of parameters values that minimize the cost function towards a local minimum or optimal accuracy

- the key assignment to solving a task presented to a neural network will be:
  - to **adjust the values of the weights and biases** in a manner that approximates or best represents the dataset

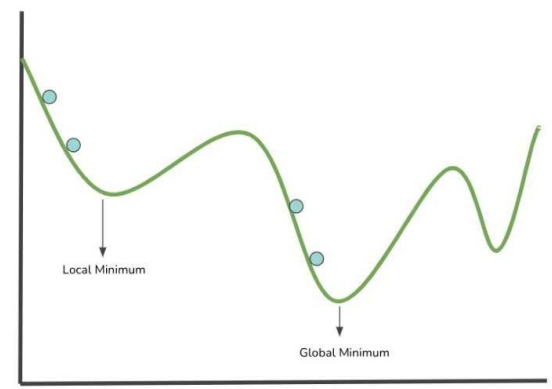


A gradient

- A measurement that quantifies the steepness of a line or curve
- Quantifies and demonstrate the direction of the ascent or descent of a line
- The magnitude and direction of the weight update is computed by taking a step in the opposite direction of the cost gradient

The aim of backprop

- is to distribute the total error back to the network
- so as to update the weights in order to minimize the cost function (loss)
- The weights are **updated after each epoch** (pass over the training dataset)
- in such a way that when the next forward pass utilizes the updated weights, the total error will be reduced by a certain margin (until the minima is reached)



Loss Functions

Quantify the error between model predictions and ground-truth.
Computed over all *N* samples in your dataset.
The loss depends on your model weights, since these determine your predictions.

Examples:

Mean-Squared Error
Used for regression

predicted value

actual value

$$J(\boldsymbol{W})_{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Categorical Cross-Entropy
Used for multi-class classification

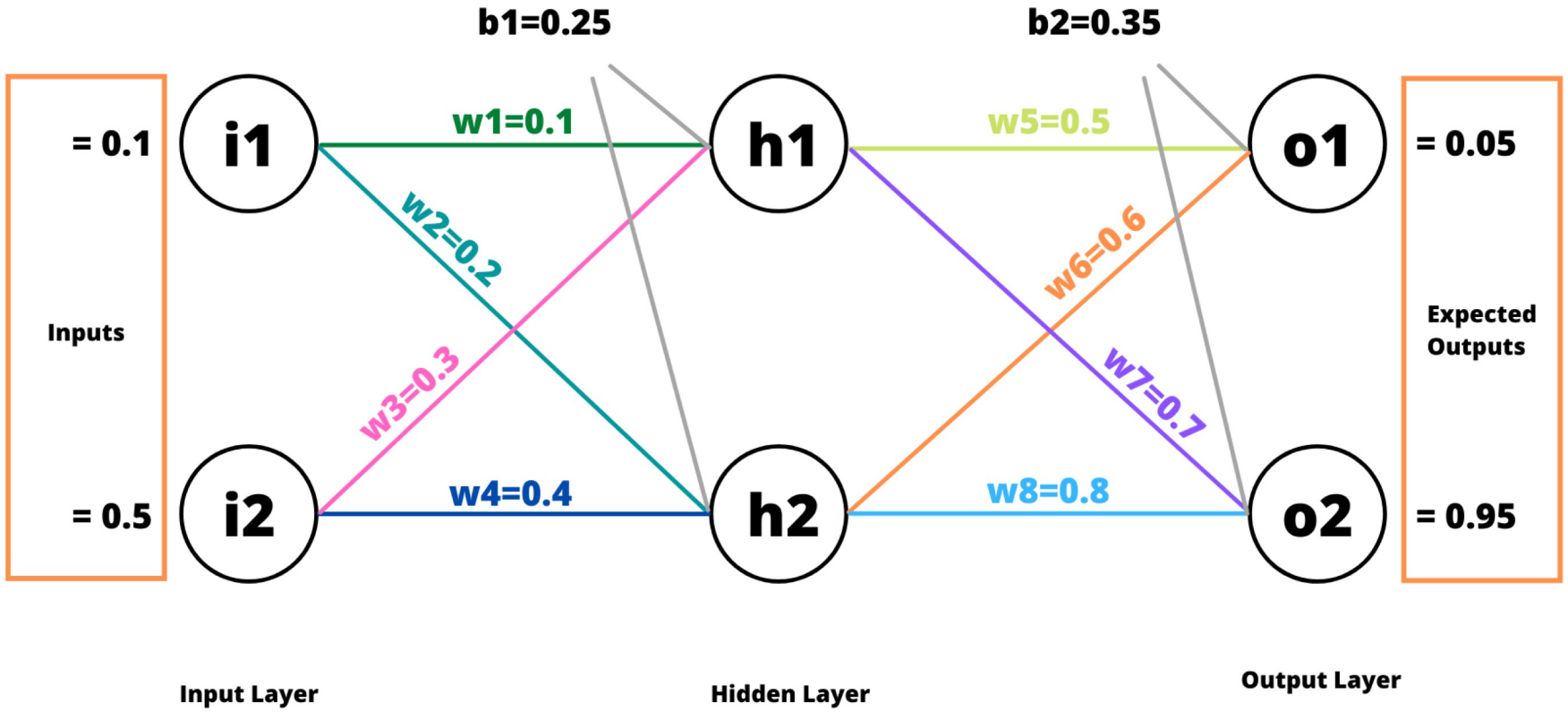
actual probability…

predicted probability…

$$J(\boldsymbol{W})_{\text{CCE}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}\hat{y}_i^{(k)}\,log(p_i^{(k)})$$

… of belonging to the *k*th class.

Exampl
e


b1=0.25
b2=0.35
= 0.1
i1
w1=0.1
h1
w5=0.5
o1
= 0.05
w2=0.2
w6=0.6
Inputs
Expected
Outputs
w3=0.3
w7=0.7
= 0.5
i2
w4=0.4
h2
w8=0.8
o2
= 0.95
Input Layer
Hidden Layer
Output Layer

Output O1:

$$sum_{o1} = output_{h1} * w_5 + output_{h2} * w_6 + b_2 = 1.01977$$

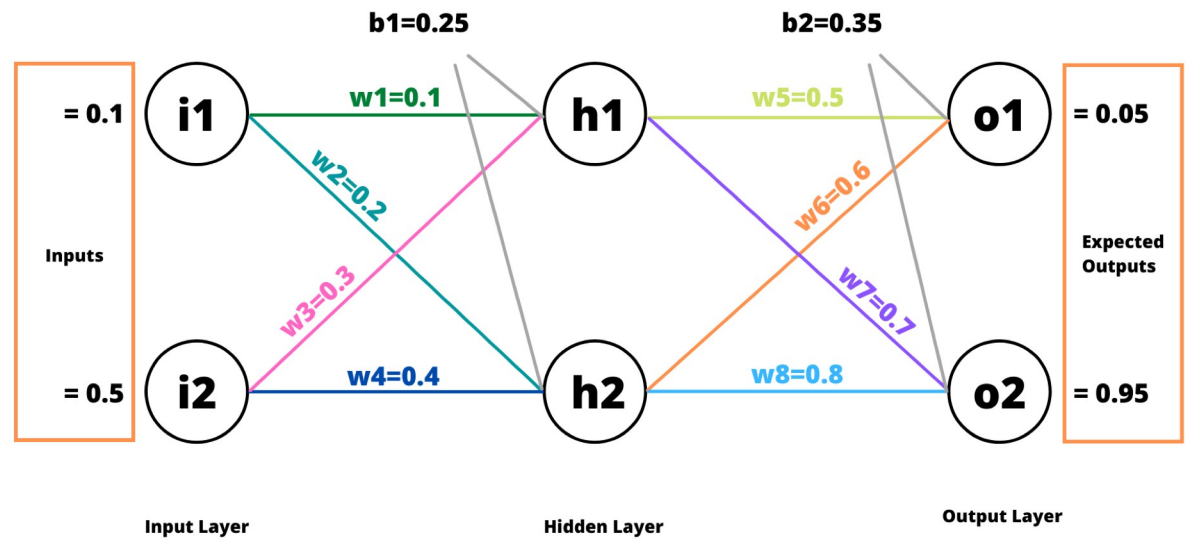
$$output_{o1} = \frac{1}{1 + e^{-sum_{o1}}} = 0.73492$$

Given expected value for O1 being 0.05, Error for O1:

$$E_1 = \frac{1}{2}(0.05 - 0.73492)^2 = 0.23456$$

Total Error:

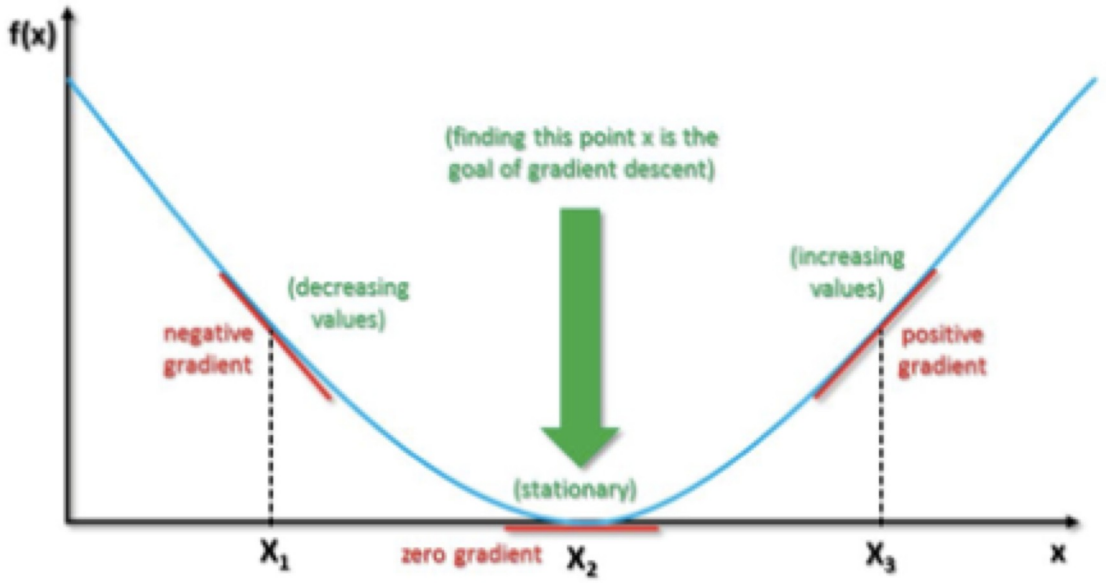
$$E_{total} = E_1 + E_2 = 0.24908$$

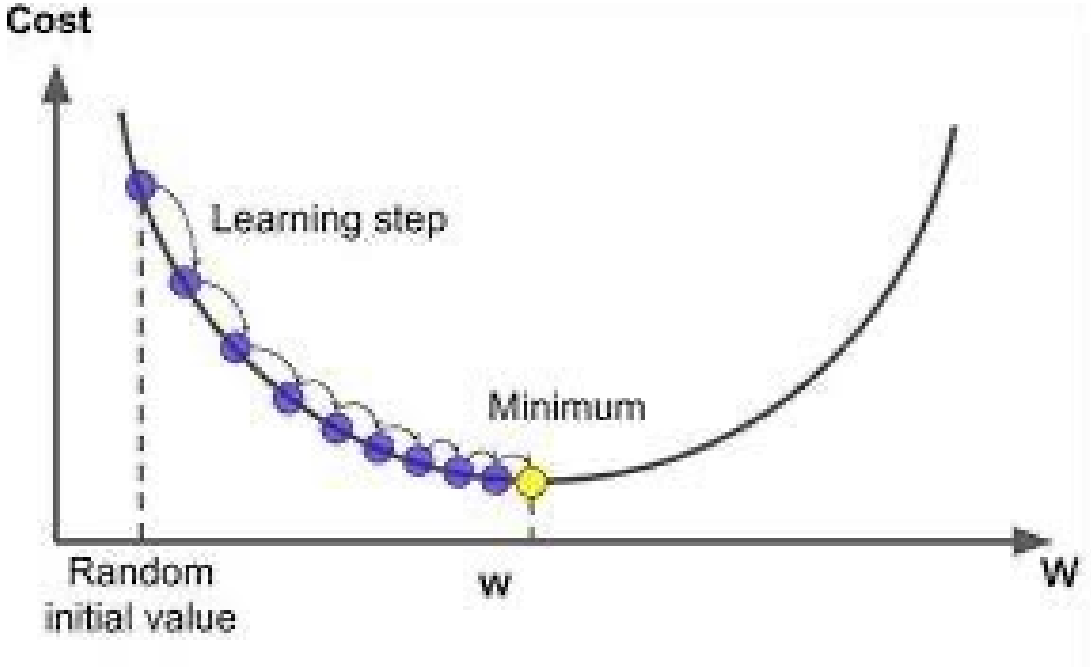


Contribution of w5 on E1 and chain rule

- E1 is affected by output O1
- output O1 is affected by Sum O1
- Sum o1 is affected by w5

$$\frac{\partial E_{total}}{\partial w5} = \frac{\partial E_{total}}{\partial output_{o1}} * \frac{\partial output_{o1}}{\partial sum_{o1}} * \frac{\partial sum_{o1}}{\partial w5}$$

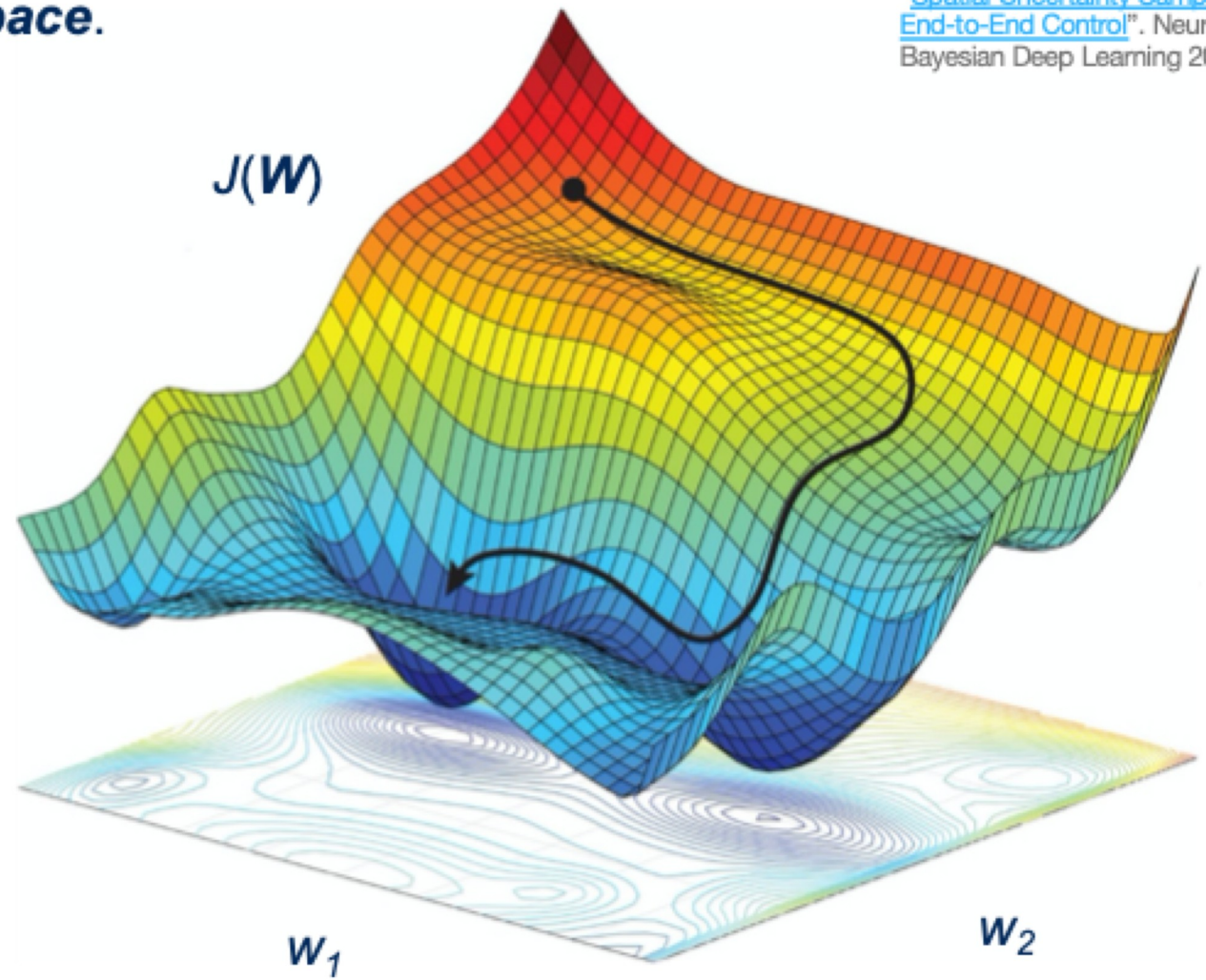




Optimizing the Loss

Goal: find the model weights $\boldsymbol{W}$ that minimize the loss function $J(\boldsymbol{W})$.
Recall: $J(\boldsymbol{W})$ is just a surface in ***weight space***.

Image Source: A. Amini et al. "Spatial Uncertainty Sampling for End-to-End Control". NeurIPS Bayesian Deep Learning 2018



Gradient Descent Algorithm:

1. Randomly initialize weights: $\boldsymbol{W}$
2. Compute gradient: $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
3. Update weights: $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Repeat steps 2-3 until convergence.

Challenges when Training DNNs

Optimizing deep neural nets in practice can be difficult:

1. Deep networks can suffer from ‘vanishing’ or ‘exploding’ gradients during backpropagation.

2. Loss landscapes are not as ‘clean’ as those ones we have shown. You can easily get stuck in a local minimum, or take a long time to converge.

3. Neural networks are prone to overfitting your data. They may perform well on your training dataset, but generalize poorly to unseen data.

4. Performance also depends on the appropriate choice of **hyperparameters**, of which **neural networks have many**, e.g. number of neurons per layer, the number of layers, layer types, activation function types, regularization, etc.

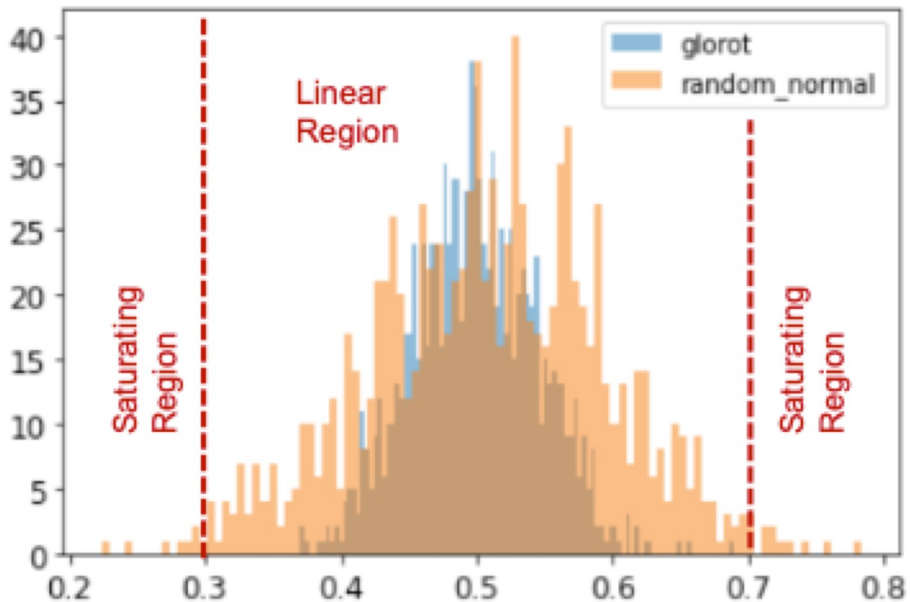
Module 2 – Section 3

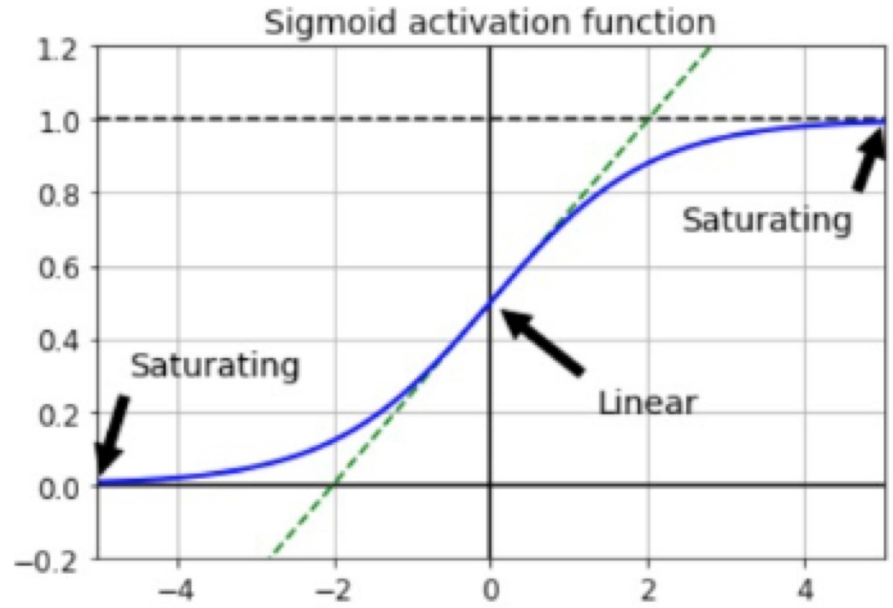
Model Tuning

Supplemental Content Only
See Jupyter Notebook for Core Content

Weight Initialization

Histogram of **Sigmoid** activation values over 1000 random weight initializations

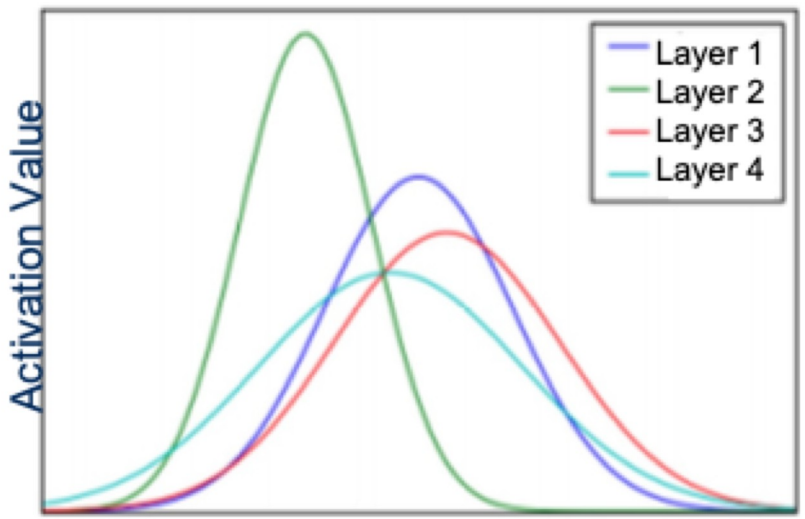




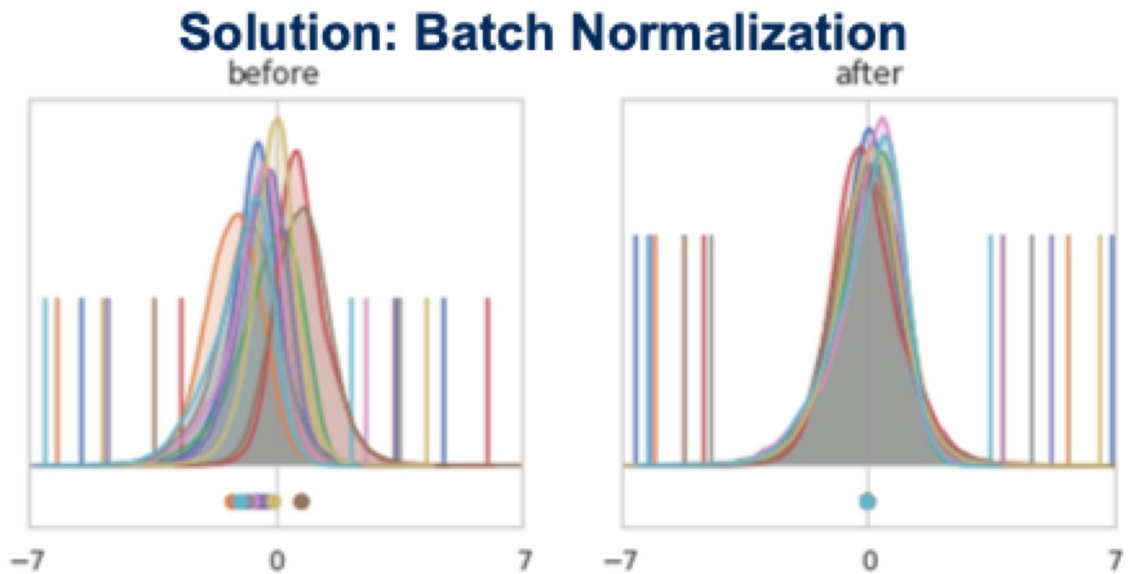
Remember: you will have hundreds of thousands of neurons or more, and therefore hundreds of thousands of activations. Ideally you want them **all** to be initialized in the non-saturating range.

Internal Covariate Shift

We want each layer to have a stable distribution of input activations. Furthermore, we want these in a range where we avoid activation function saturation. However, during training, the mean and standard deviation of activations from each neuron can shift. This can push us towards saturation, and slows training by forcing each layer to adapt to new input distributions each iteration.



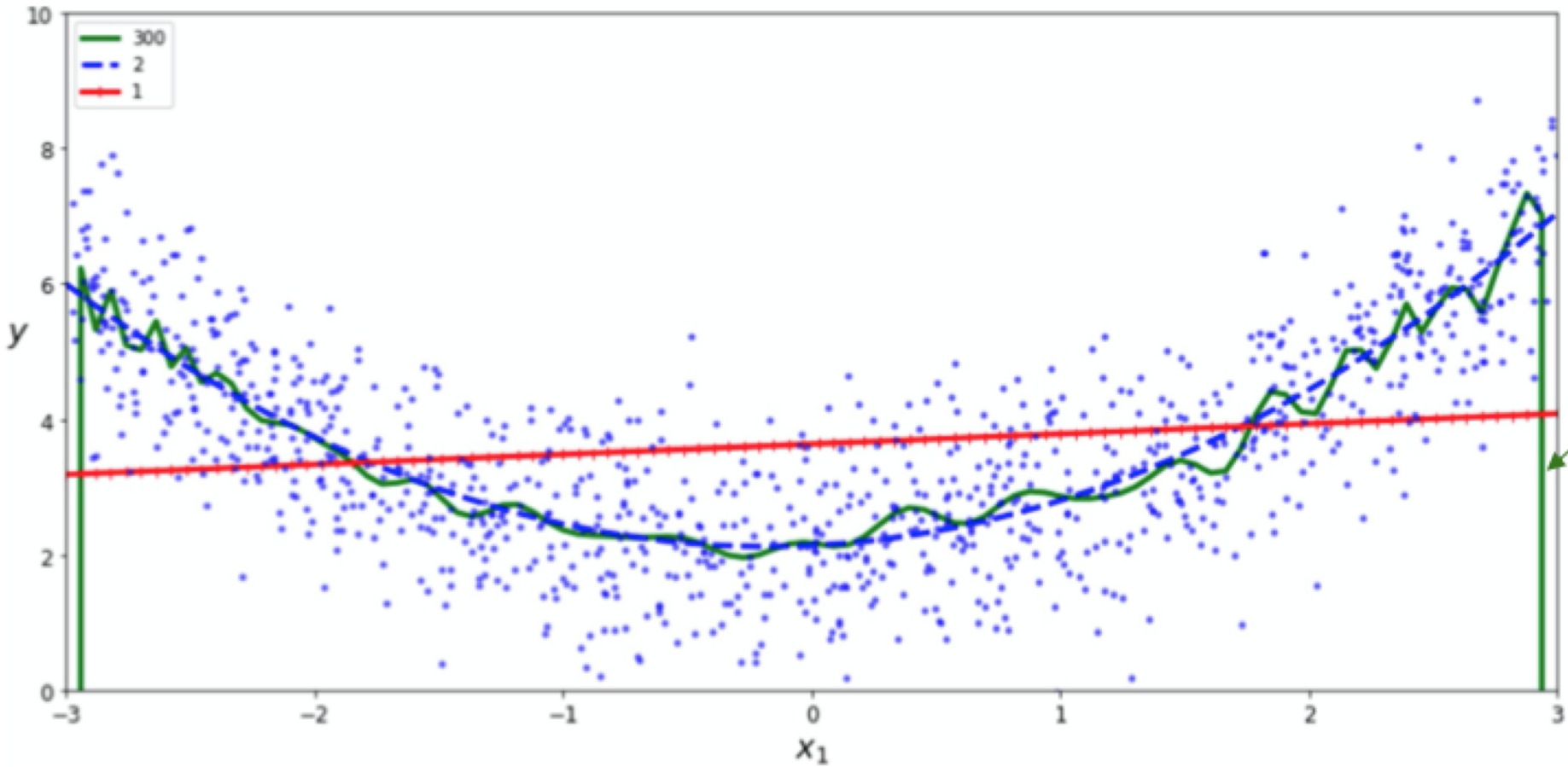
Probability Density (Histogram)



Source, this post by David C Page:
https://colab.research.google.com/github/davidcpage/cifar10-fast/blob/master/batch_norm_post.ipynb

Overfitting and Underfitting

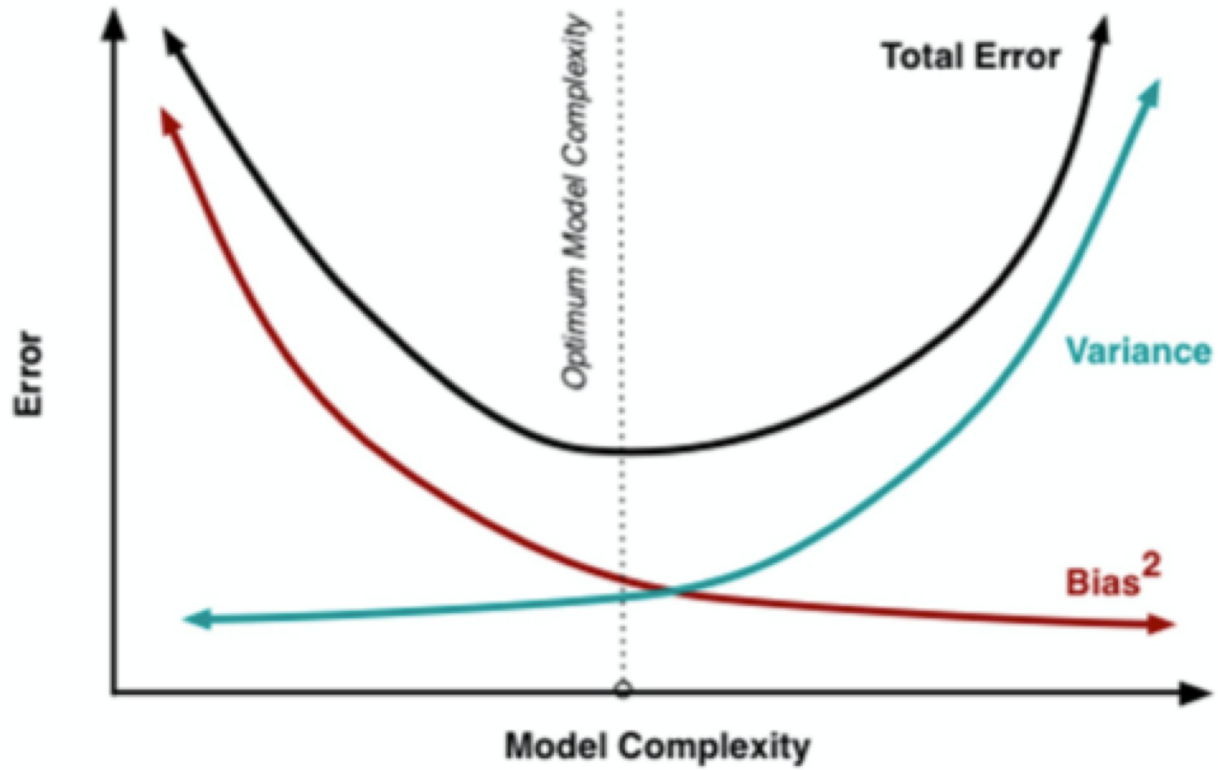
Example: polynomial features; lower degrees = underfitting, higher degrees = overfitting



Overfitting: predictions would fail utterly for values beyond 3 or -3.

Bias-Variance Tradeoff

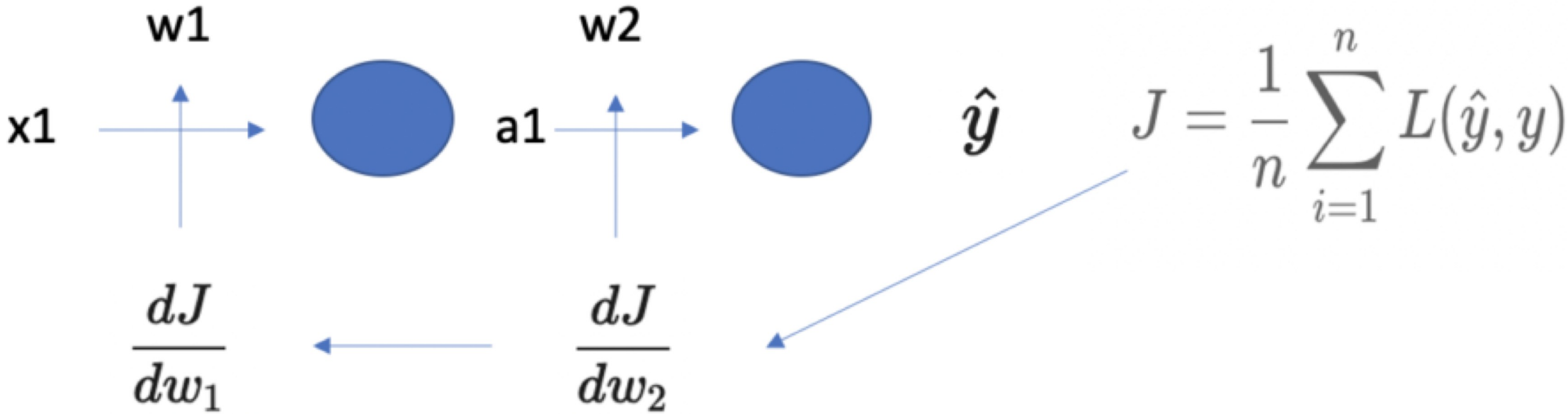
By tuning the model complexity (aka “capacity”), we can reduce error due to overfitting.
Decreasing the number of neurons in a hidden layer is one example of reducing capacity.



Source: Cornell University
https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote12.html

Vanishing and exploding gradients problem

$$z_1 = w_1 x_1 \rightarrow a_1 = \sigma(z_1) \rightarrow z_2 = w_2 a_1 \rightarrow \hat{y} = \sigma(z_2) \rightarrow J = \frac{1}{n}\sum_{i=1}^{n} L(\hat{y}, y)$$



Vanishing gradients problem

- Gradients often get smaller and smaller as the algorithm progresses down to the lower layers - i.e. closer to the inputs
- The Gradient Descent update leaves the lower layers' connection weights virtually unchanged
- Result:
  - training never converges to a good solution
  - prevent the network from learning long-term dependencies

Exploding gradients problem

- the gradients can grow bigger and bigger until layers get insanely large weight updates and the algorithm diverges.
- An issue in RNNs - recurrent neural networks
- Solution: Gradient Clipping

Unstable gradients:

- Different layers may **learn at widely different speeds**

Initialization	Activation functions	σ^2 (Normal)
Glorot	None, tanh, logistic, softmax	$1 / fan_{avg}$
He	ReLU and variants	$2 / fan_{in}$
LeCun	SELU	$1 / fan_{in}$

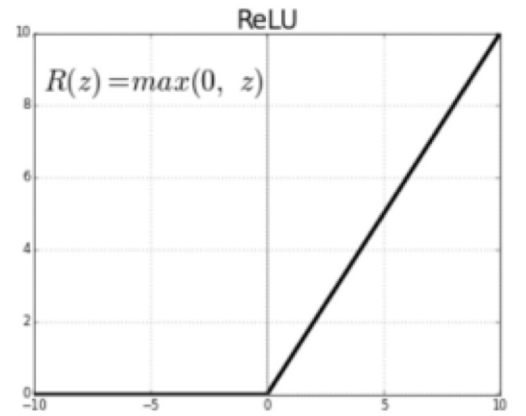
Solutions:

- Proper Weight Initialization
- Using Non-saturating Activation Functions
- Batch normalization
- Gradient Clipping

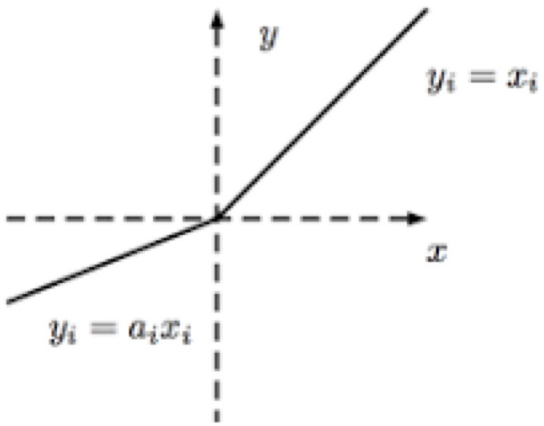
1. Proper Weight Initialization
    - The random normal weight initialization technique
    - a normal distribution with a mean of 0 and a standard deviation of 1

Using Non-saturating Activation Functions (ReLU)

ReLU
(Rectified Linear Unit)



LReLU
(Leaky ReLU)



Solutions

3. Batch Normalization

- stabilizing the learning process (and making it faster)
- normalization of the layers' inputs by re-centering and re-scaling
- Address the internal covariate shift problem

internal covariate shift problem

as the training process progresses from epoch to epoch, the **distribution of each layer's inputs can change significantly as the parameters of the previous layers change**

https://arxiv.org/pdf/1805.11604.pdf

- Batch normalization layers are added in the model just before (or after) the activation function of each layer

Process:

Ultimately batch normalization lets the model learn the **optimal scale and mean of the inputs for each layer**

Two main steps:

1. **Zero-centering the inputs:**
    - In order to zero-center and normalize the inputs
        - **the algorithm needs to estimate the mean and standard deviation of the inputs**
    - evaluate the **mean** and **standard deviation** of the inputs over the **current mini-batch** (hence the name “Batch Normalization”).
2. **Scaling and shifting:**
    - The zero-centered inputs are then **scaled** and **shifted** using two new **learnable** parameters per layer (one for scaling and one for shifting)

Disadvantages of Batch Normalization:

- Batch Normalization does add some complexity to the model
- There is a runtime penalty: the neural network makes **slower predictions due to the extra computations required at each layer**.
  - Fortunately, it's often possible to **fuse the BN layer with the previous layer, after training, thereby avoiding the runtime penalty**
  - This is done by updating the previous layer's weights and biases so that it directly produces outputs of the appropriate scale and offset.

4. Gradient Clipping

- This optimizer will clip every component of the gradient vector to a value between –1.0 and 1.0
  - i.a. All the partial derivatives of the loss with respect to each trainable parameter
- The threshold is a hyperparameter we can tune
- most often used in recurrent neural networks (RNNs)

```
model.compile(loss="sparse_categorical_crossentropy",
              optimizer=keras.optimizers.SGD(lr=1e-3),
              metrics=["accuracy"])
```

```
optimizer = keras.optimizers.SGD(clipvalue=1.0)
```

Optimizer

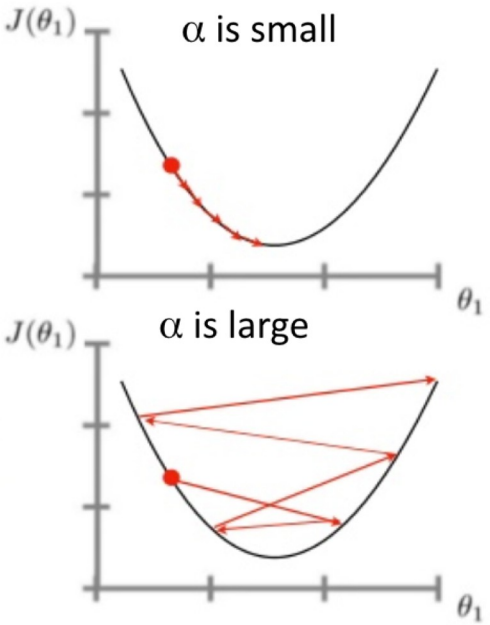
An optimizer is a function or an algorithm that adjusts the attributes of the neural network, such as weights and learning rates. Thus, it helps in reducing the overall loss and improving accuracy

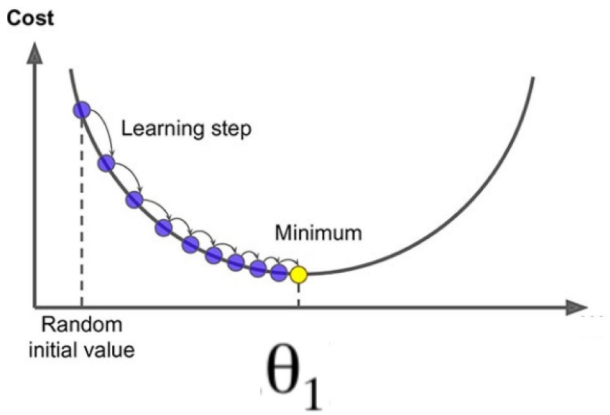
repeat until convergence {

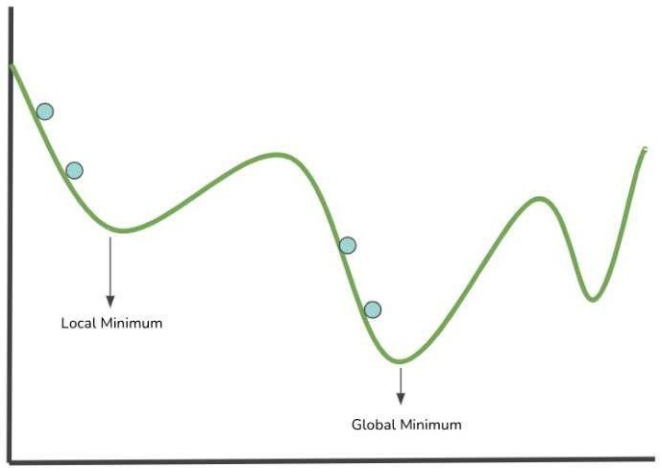
$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

(for $j = 1$ and $j = 0$)

}







Training a very large deep neural network can therefore be painfully slow

Different ways to speed up training

1. applying a **good initialization strategy** for the connection weights,
2. using a **good activation function**,
3. using **Batch Normalization**

4. using a **faster optimizer than the regular Gradient Descent optimizer:**

- **momentum optimization**
  - a widely-used strategy for accelerating the convergence of gradient-based optimization techniques
  - Momentum was designed to speed up learning in directions of low curvature, without becoming unstable in directions of high curvature

- **Nesterov Accelerated Gradient**
  - is an extension of momentum
  - involves calculating the decaying moving average of the gradients of projected positions in the search space rather than the actual positions themselves

- **AdaGrad** (Adaptive Gradient Algorithm)
  - The idea is to use different learning rates for each parameter base on iteration
  - different learning rates:
    - Because the learning rate for sparse features needs to be higher compare to the dense features parameter because the frequency of occurrence of sparse features is lower
  - AdaGrad adds element-wise scaling of the gradient based on the historical sum of squares in each dimension

- **RMSProp**
  - Similar to AdaGrad
  - RMSprop keep estimate of squared gradients, but instead of letting that estimate continually accumulate over training, it keep a moving average of it

- **Adam (Adaptive Movement Estimation) and Nadam (Nesterov-accelerated Adaptive Moment Estimation) optimization**
  - Adam: uses a separate step size for each input variable
  - Nadam: is an extension of the Adam algorithm that incorporates Nesterov momentum and can result in better performance of the optimization algorithm

Table 11-2. Optimizer comparison

Class	Convergence speed	Convergence quality
`SGD`	*	***
`SGD(momentum=...)`	**	***
`SGD(momentum=..., nesterov=True)`	**	***
`Adagrad`	***	* (stops too early)
`RMSprop`	***	** or ***
`Adam`	***	** or ***
`Nadam`	***	** or ***
`AdaMax`	***	** or ***

learning rate

- Rather than a constant learning rate, better to start with a high learning rate and then reduce it once it stops making fast progress

- different strategies to reduce the learning rate during training **(learning schedules)**

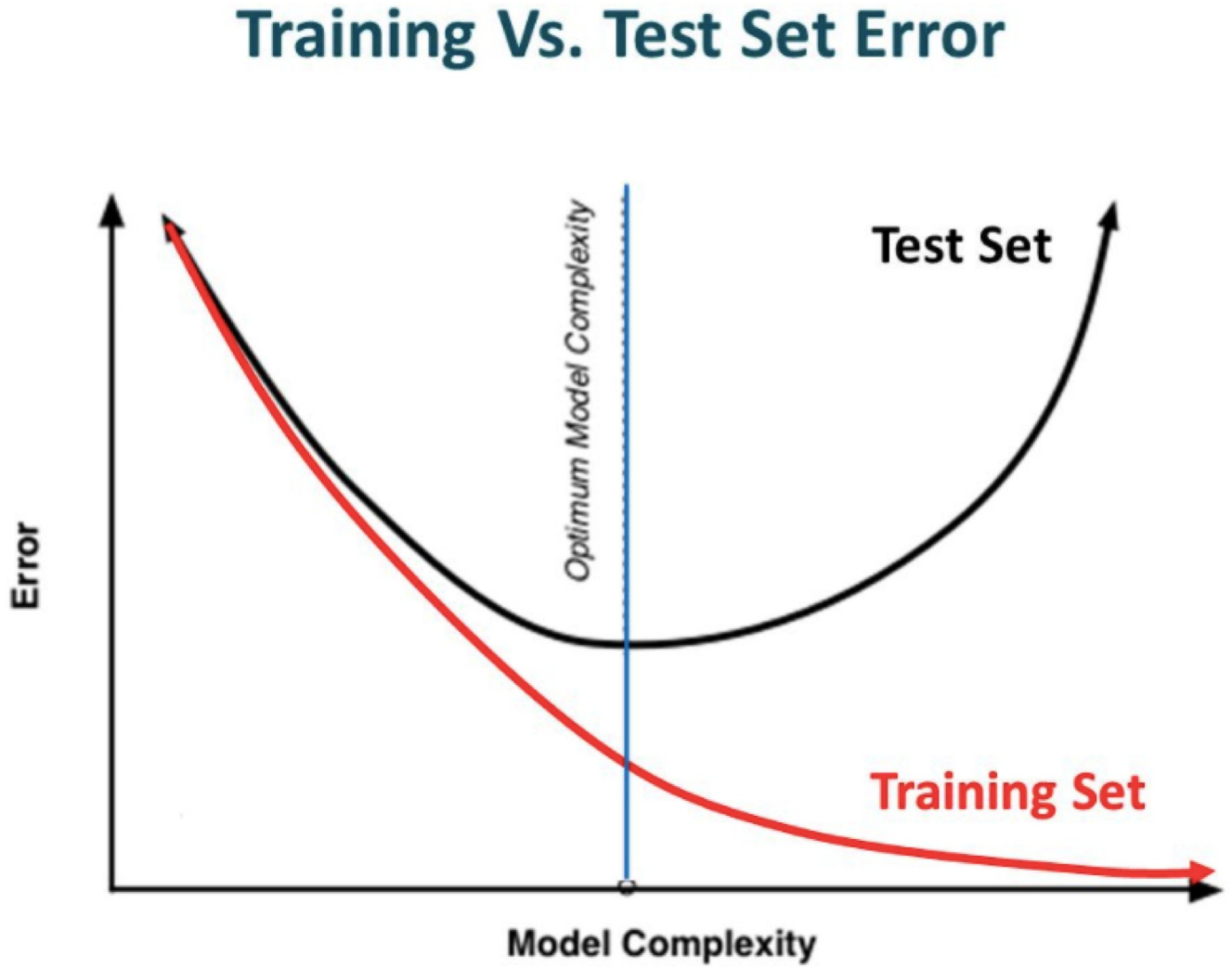
- **Exponential scheduling:**
  - Set the learning rate to a function of the iteration number
  - E.g. The learning rate drop by a factor of 10 every s steps

- **Power scheduling:**
  - Set the learning rate to a function of the iteration number
  - As you can see, this schedule first drops quickly, then more and more slowly

- **Piecewise constant learning rate:**
  - The learning rate reduces based on predefined update steps

- **Performance scheduling:**
  - Measure the validation error every N steps and reduce the learning rate by a factor when the error stops dropping

Regularization



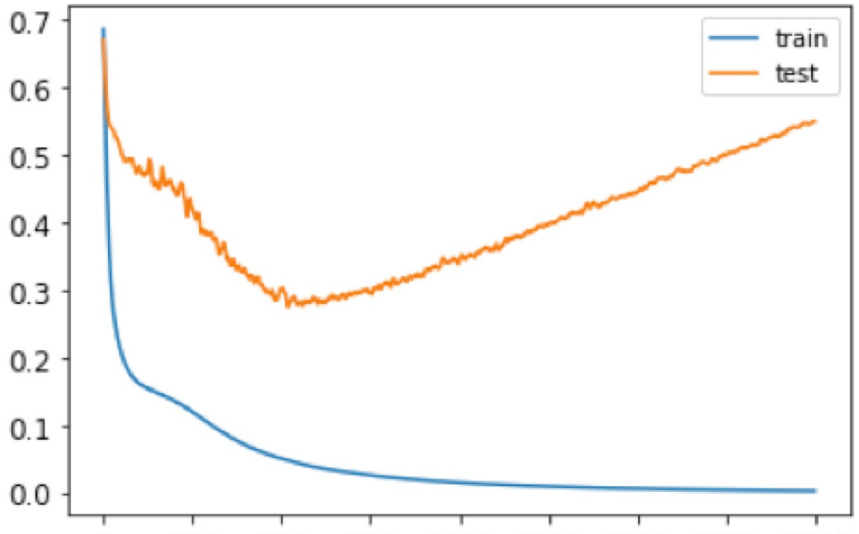
Regularization

1. **Early Stopping**

- The choice of the **number of training epochs** to use
- Too many epochs can lead to overfitting of the training dataset
- Too few epochs may result in an underfit model
- Early stopping is a method that allows you to specify an arbitrary large number of training epochs and stop training once the model performance stops improving on a hold out validation dataset

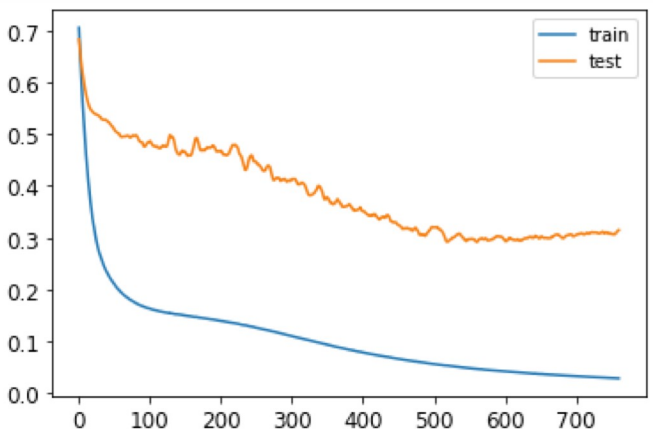
```
from tensorflow.keras.callbacks import EarlyStopping
EarlyStopping(monitor='val_loss',
              min_delta=0,
              patience=0,
              verbose=0,
              mode='auto',
              baseline=None,
              restore_best_weights=False)
```

```
history = model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=2000, verbose=0)
```




```
# Add early stopping
es = EarlyStopping(monitor='val_loss', mode='min', patience=200, restore_best_weights=True, verbose=1)

# fit model
history = model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=2000, callbacks=[es], verbose=0)
```



Choosing the best model:

restore_best_weights

- When set to true our model automatically reverts to the best performing weights

ModelCheckpoint

- If you are training your model for many hours and want to automatically save the best model when a global minimum is reached
- Add a ModelCheckpoint callback into the fit function

```
mc = ModelCheckpoint('best_model.h5', monitor='val_acc', mode='max', verbose=1, save_best_only=True)
```

L1 and L2 Regularization

2. L1 and L2 Regularization

The capacity of the models like neural networks, linear or logistic regression is limited by adding a parameter norm penalty Ω(θ) to the objective function J

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta)$$

α *lies within [0, ∞) is a hyperparameter that weights the relative contribution of a norm penalty term, Ω, pertinent to the standard objective function J*

L2:

- weight decay or Ride Regression
- most common type of all regularization techniques
- Regularization adds the penalty as model complexity increases

L1:

- Lasso Regression (**Least Absolute Shrinkage and Selection Operator**)
- adds **"Absolute value of magnitude"** of coefficient, as penalty term to the loss function

1. Dropout

- The most popular regularization technique for deep learning
- At every training step, every neuron (including the input neurons but excluding the output neurons) has a probability of being temporarily “dropped out”
- Meaning it will be ignored during this training step, but it may be active during the next step
- The hyperparameter representing the probability is called the dropout rate, and it is typically set to 50%
- After training, neurons do NOT get dropped anymore

Max-norm Regularization

- constrains the weight sizes of the incoming connections in each layer
- The weight vector associated with each neuron is forced to have an $\ell 2$ norm of at most a Hyperparameter
- If this constraint is not satisfied, the weight vector is replaced by the unit vector in the same direction that has been scaled by the hyperparameter

https://keras.io/api/layers/constraints/

```
from tensorflow.keras.constraints import MaxNorm
...
mn = MaxNorm(max_value=3)
model.add(Dense(32, activation='relu', kernel_constraint=mn))
```

Data Augmentation strategies like:

- altering the model size (e.g. adding layers or units to each layer)
- adjusting the learning rate, etc.
- quality of the training data may be the issue (too noisy, wrong inputs, etc)
- collecting "better" data may be more of an immediate need that simply collecting more data
- if the performance on the training set is good, but the performance on the test set is poor, gathering additional data (especially labelled data) should be highly effective in improving performance
- **Data Augmentation** involves generating new training instances from existing ones, artificially boosting the size of your training set
  - this technique could likely help reduce overfitting
  - particularly common for audio and image classification problems
  - Adding Gaussian white noise to expand a dataset

Hyperparameters optimization

A hyperparameter for a model typically has the following characteristics:

1. It is external to the model itself, and can not be estimated by the model itself
2. It is set before the model begins learning
3. It can be tuned to improve performance of the model

Hyperparameters can be selected either manually or in an automated way

- Manually -> need a good understanding of what they do and how they affect model performance
- Automatic -> require additional computational time as they are often found through permutation

Automated Tuning

Hyperparameter	Increases Capacity When	Reason	Caveats
Number of hidden units	Increased	Increasing the number of hidden units increases the representational capacity of the model	Increasing the number of hidden units increases both the time and memory cost of essentially every operation on the model
Learning rate	Tuned optimally	An improper learning rate, whether too high or too low, results in a model with low effective capacity due to optimization failure	
Implicitly zero padding	Increased	Adding implicit zeros before convolution keeps the representation size large.	Increases time and memory cost of most operations
Weight decay coefficient	Decreased	Decreasing the weight decay coefficient frees the model parameters to become larger	
Dropout rate	Decreased	Dropping units less often gives the units more opportunities to "conspire" with each other to fit the training set	

Automated Tuning

Grid search:

- **Starting point**
- a small set of finite values are pre-selected covering the range of possible values of a given hyperparameter
- These parameter combinations are then permuted, and the model evaluated

Random search:

- A less computationally expensive alternative to a grid search
- a marginal distribution is specified for each hyperparameter (e.g. Bernouli for binary variables)
- Hyperparameter values are then sampled from each distribution without any binning or discretizing.
- As for a grid search, once the hyperparameter values have been selected, the model is evaluated
- Since multiple hyperparameter values are being modified simultaneously, random search tends to find effective solutions much more quickly than a grid search

Debugging

When model performance is poor, it can often be difficult to determine if there was a bug in the code itself, or there was improper construction of the model

unrealistically high performance may also indicate an error in the code or model setup that should be addressed.

These are important tasks to consider when debugging a neural network model:

- **Visualize the model output:**
  - Select a random subset of model predictions and manually examine for accuracy
  - If possible, identify where the model misclassified inputs and examine if this was due to a systematic error

- **Fit a small and/or well-characterized dataset:**
  - If model performance is difference than that expected, it's possible a software or implementation issue is the cause
  - It's often useful to use a simplified implementation of your model to verify that results match your expectation