Review of Machine Learning

Legal Notices and Disclaimers

This presentation is for informational purposes only. INTEL MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at intel.com.

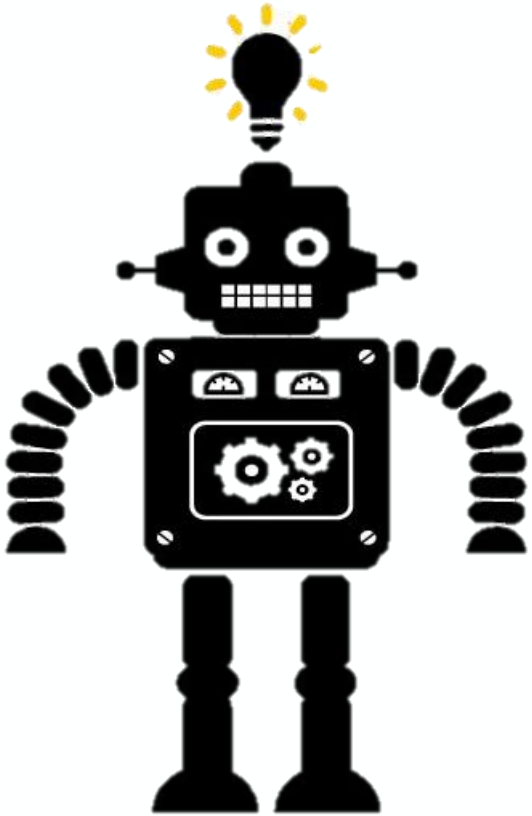
This sample source code is released under the Intel Sample Source Code License Agreement.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

What is Machine Learning?

Machine learning allows computers to learn and infer from data.



Types of Machine Learning

Supervised — data points have known outcome

Unsupervised — data points have unknown outcome

Types of Supervised Learning

Regression — outcome is continuous (numerical)

Classification — outcome is a category

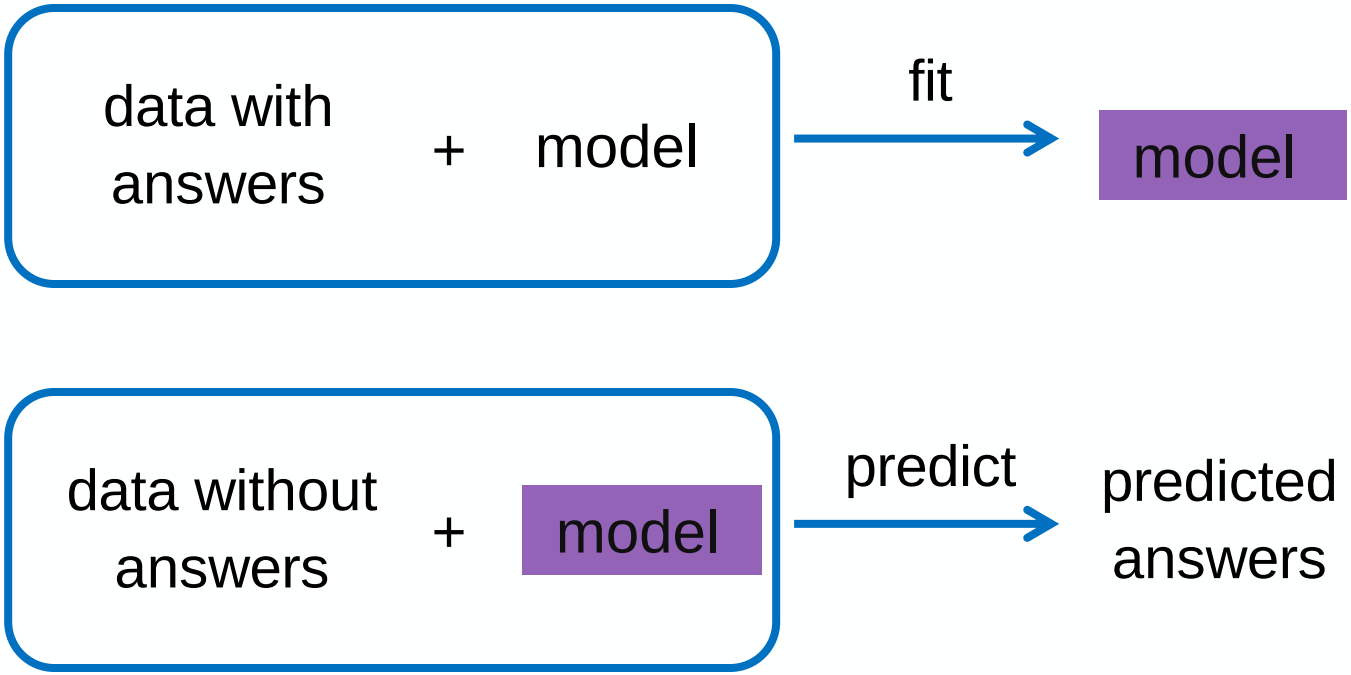
Machine Learning Vocabulary

- **Target:** predicted category or value of the data (column to predict)
- **Features**: properties of the data used for prediction (non-target columns)
- **Example:** a single data point within the data (one row)
- **Label:** the target value for a single data point

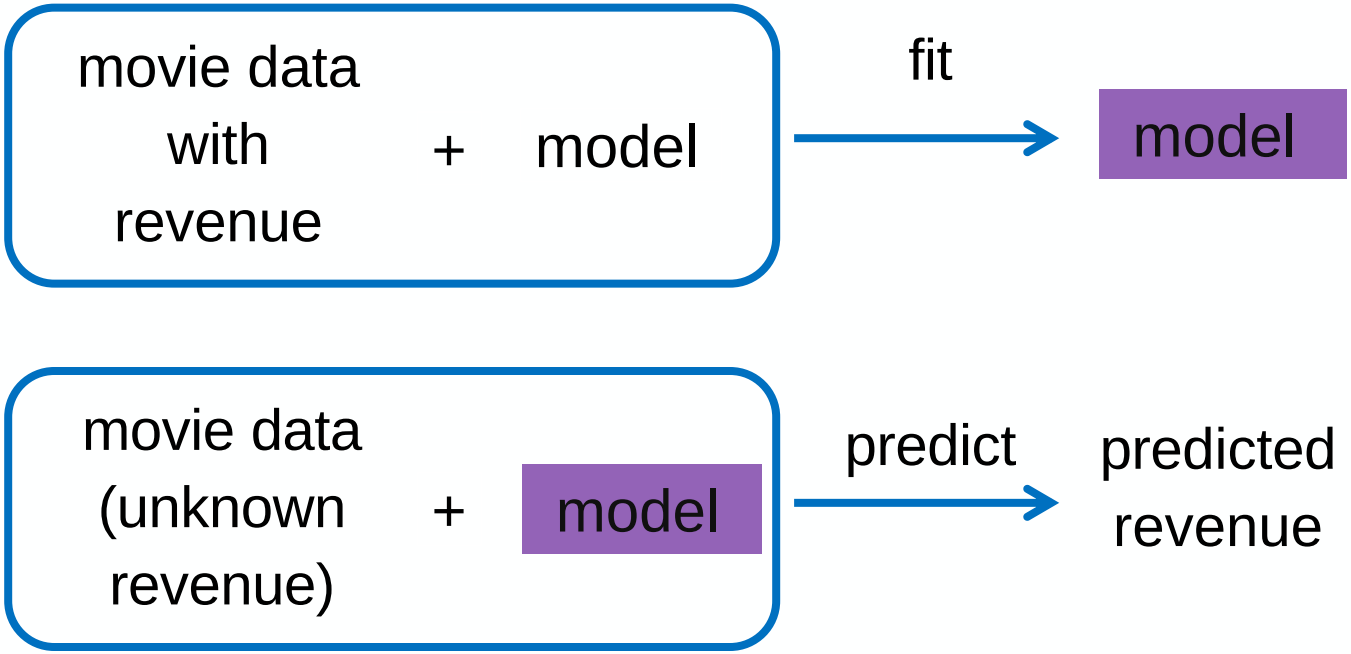
Machine Learning Vocabulary (Synonyms)

- **Target:** Response, Output, Dependent Variable, Labels
- **Features**: Predictors, Input, Independent Variables, Attributes
- **Example:** Observation, Record, Instance, Datapoint, Row
- **Label:** Answer, y-value, Category

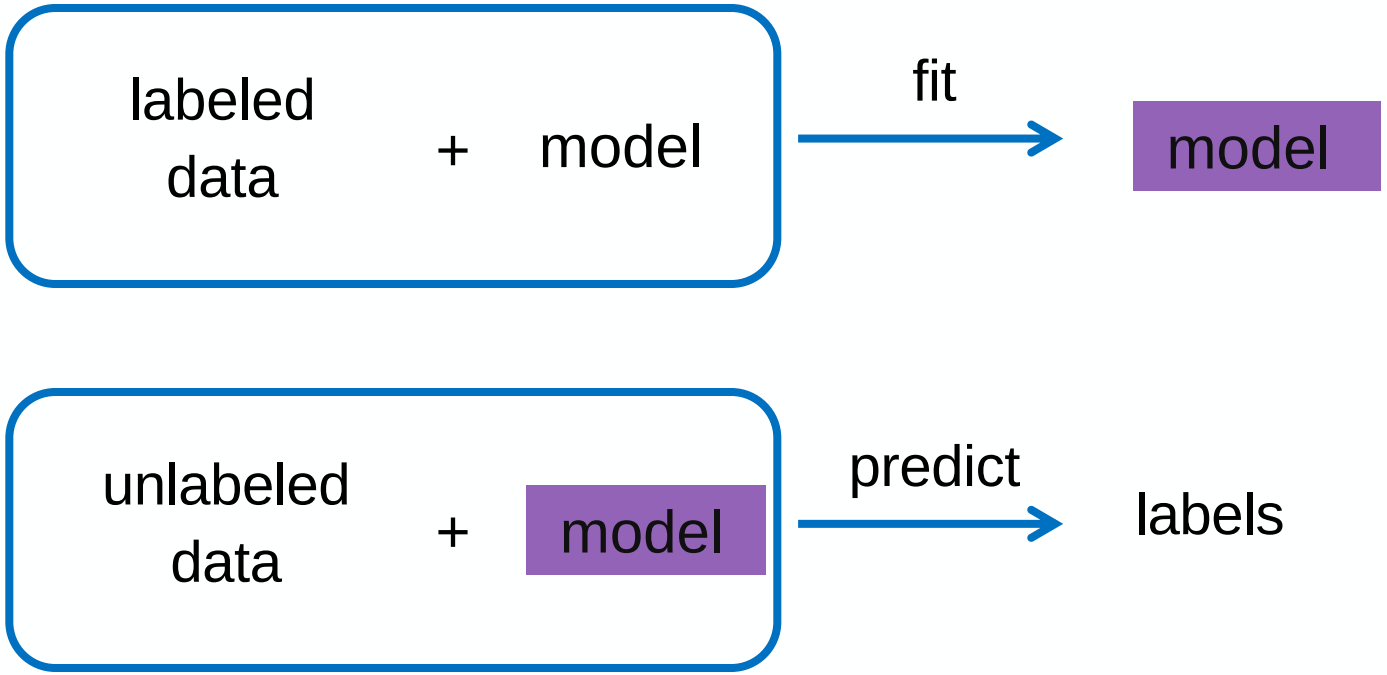
Supervised Learning Overview



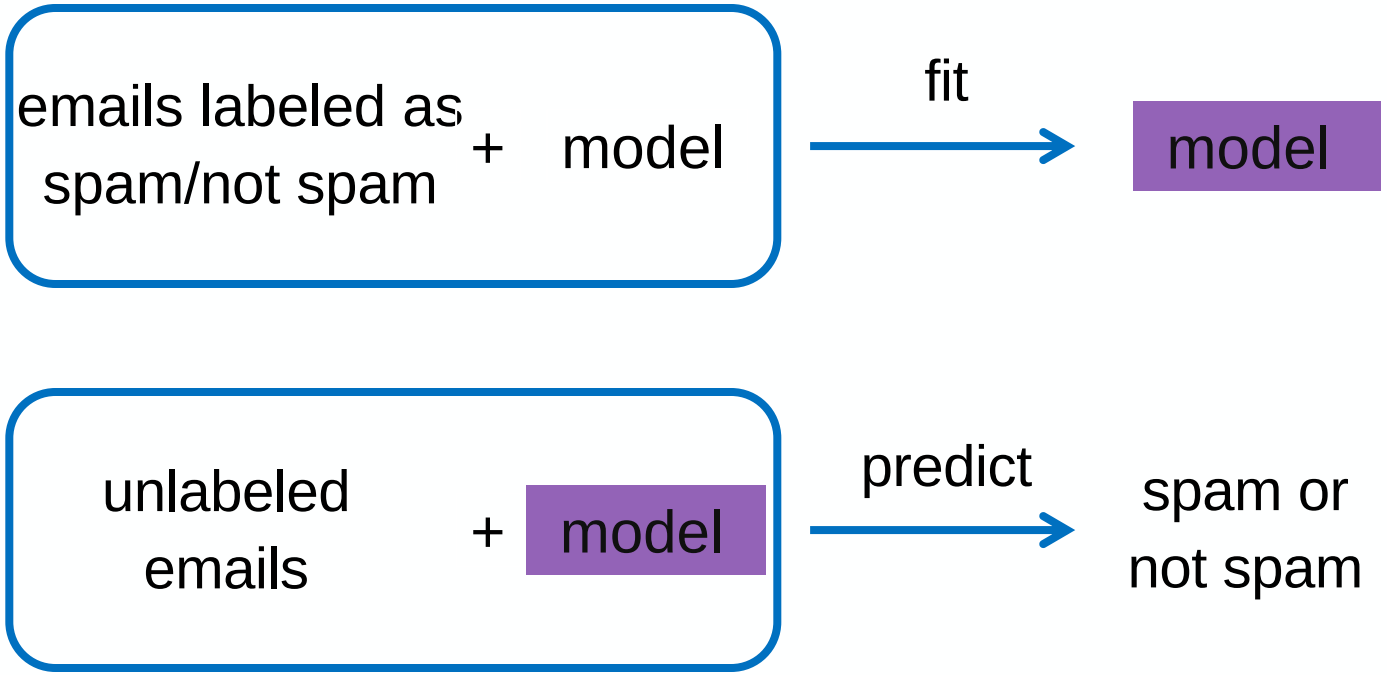
Regression: Numeric Answers



Classification: Categorical Answers



Classification: Categorical Answers



Three Types of Classification Predictions

- **Hard Prediction:** Predict a single category for each instance.
- **Ranking Prediction**: Rank the instances from most likely to least likely. (binary classification)
- **Probability Prediction:** Assign a probability distribution across the classes to each instance.

Metrics for Classification

- **Hard Prediction:** Accuracy, Precision, Recall (Sensitivity), Specificity, F1 Score
- **Ranking Prediction**: AUC (ROC), Precision-Recall Curves
- **Probability Prediction:** Log-loss (aka Cross-Entropy), Brier Score

Metrics for Regression

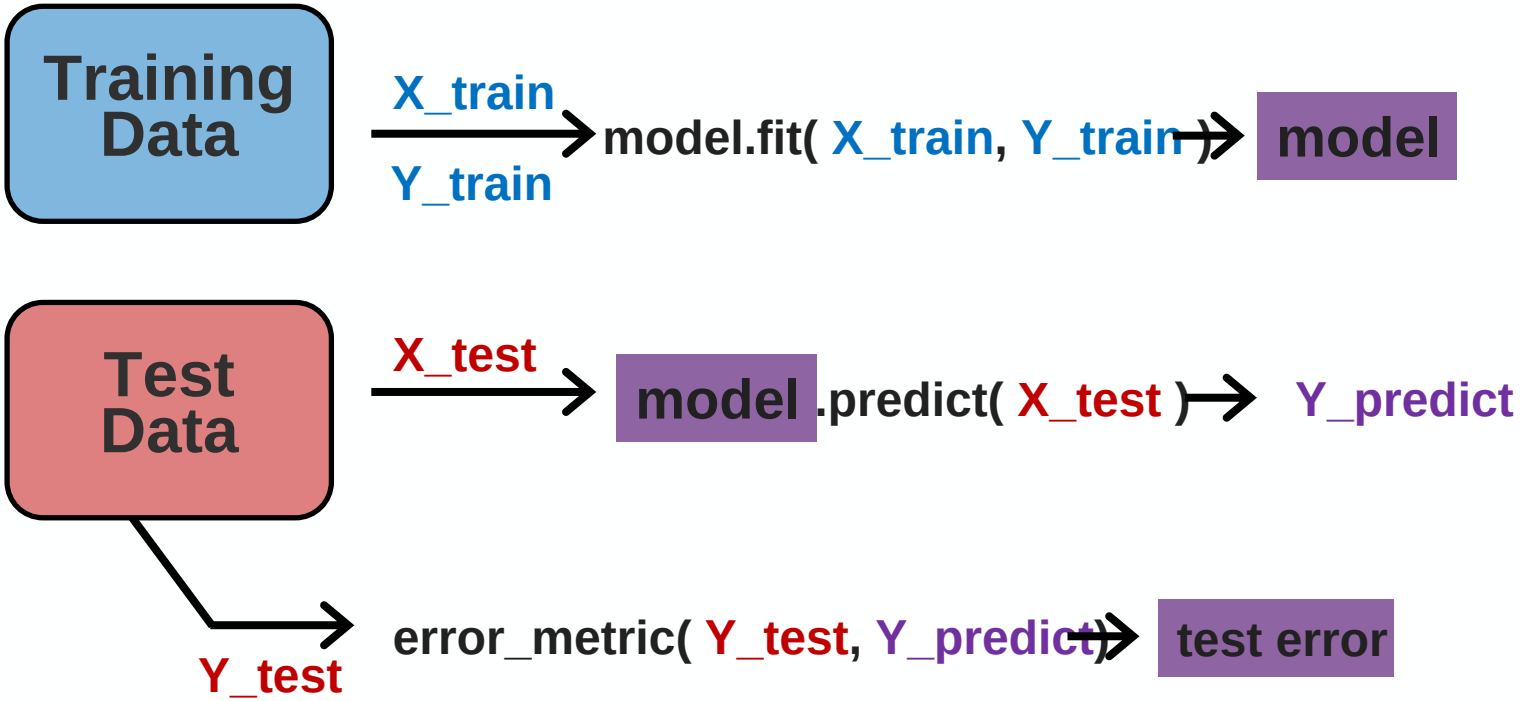
- **Root Mean Square Error (RMSE)**

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

- **Mean Absolute Deviation**

$$MAD = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Fitting Training and Test Data



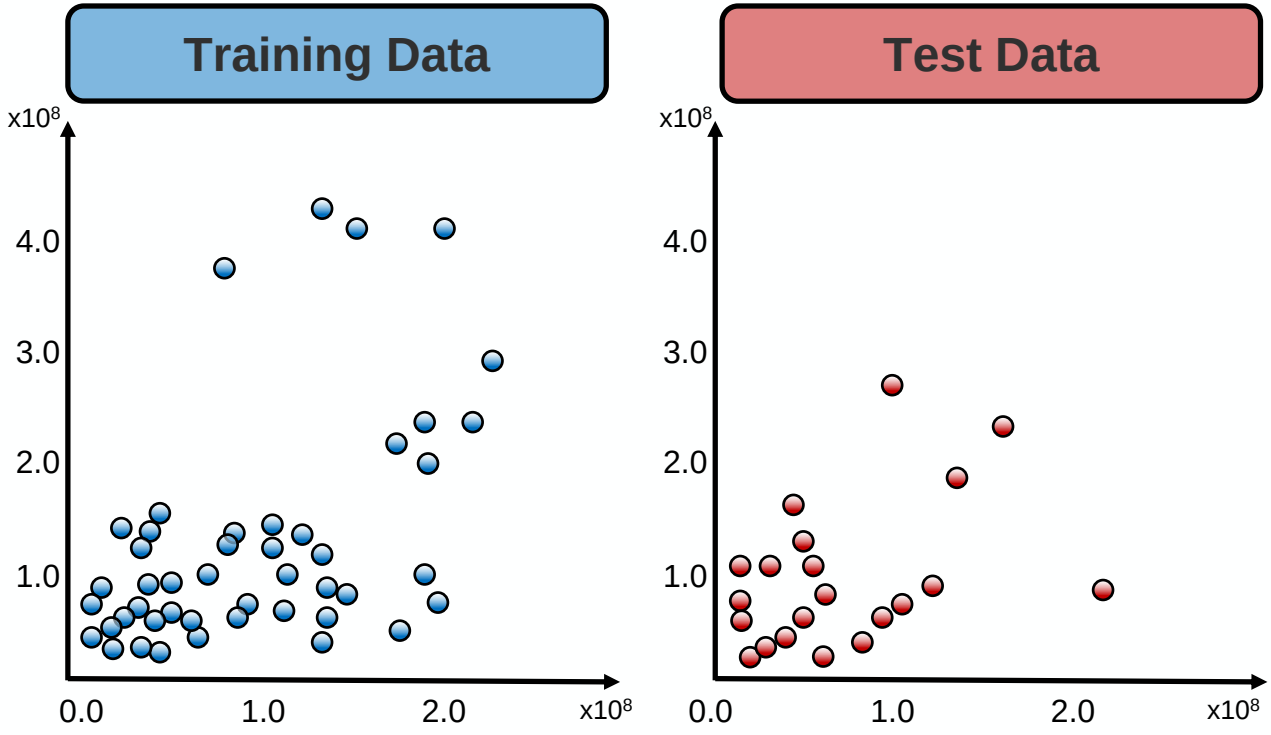
Using Training and Test Data

fit the model

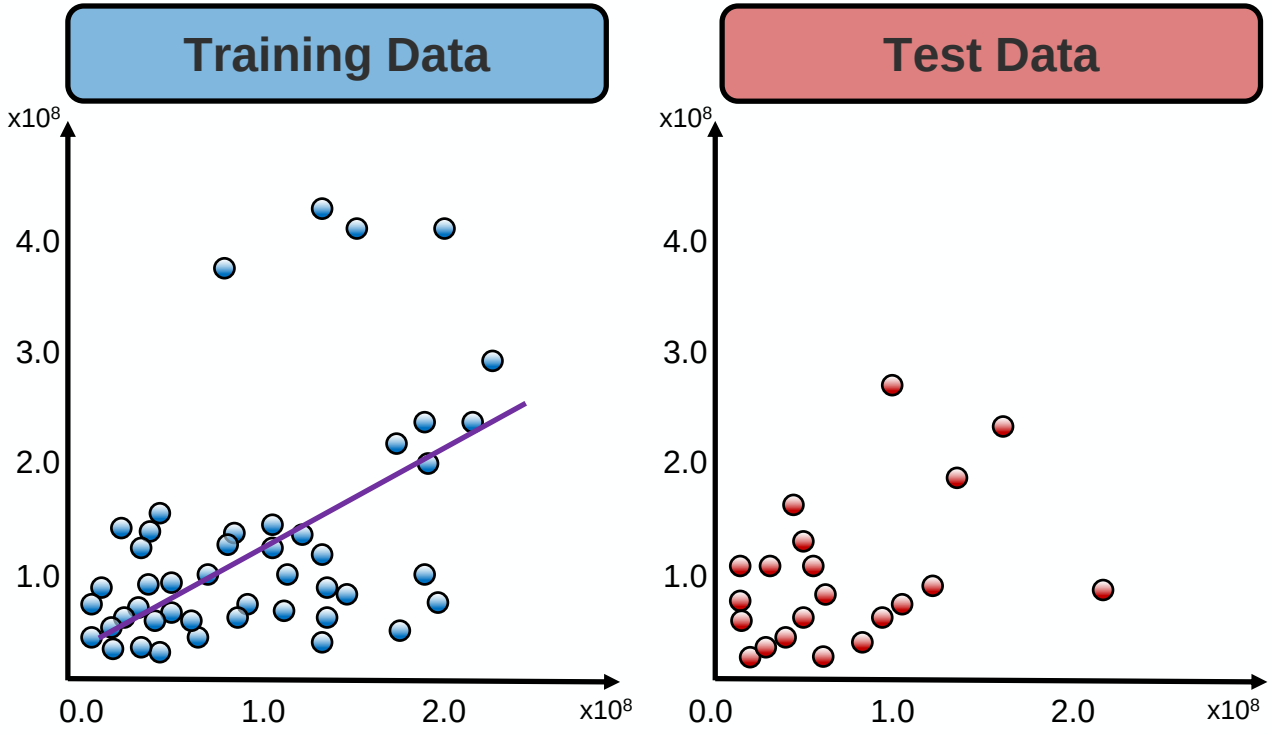
measure performance

- predict label with model
- compare with actual value
- measure error

Using Training and Test Data

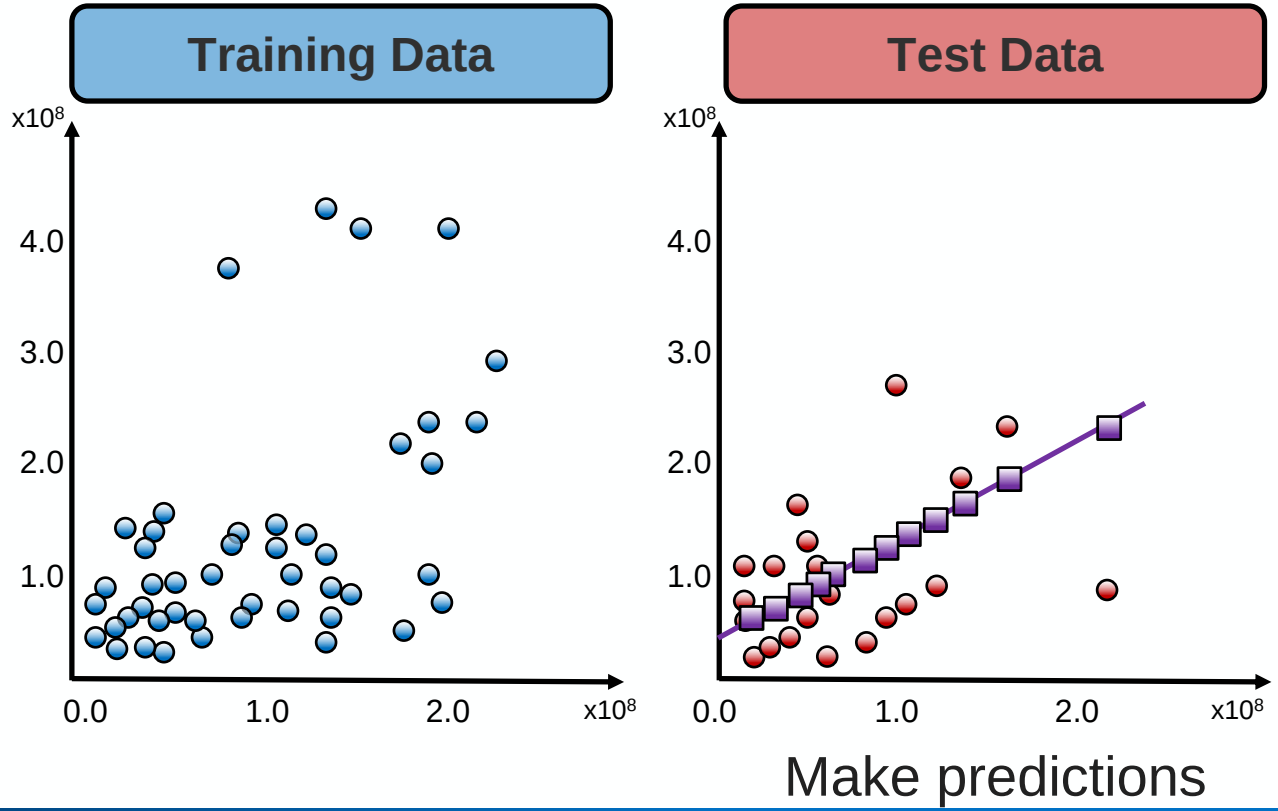


Using Training and Test Data



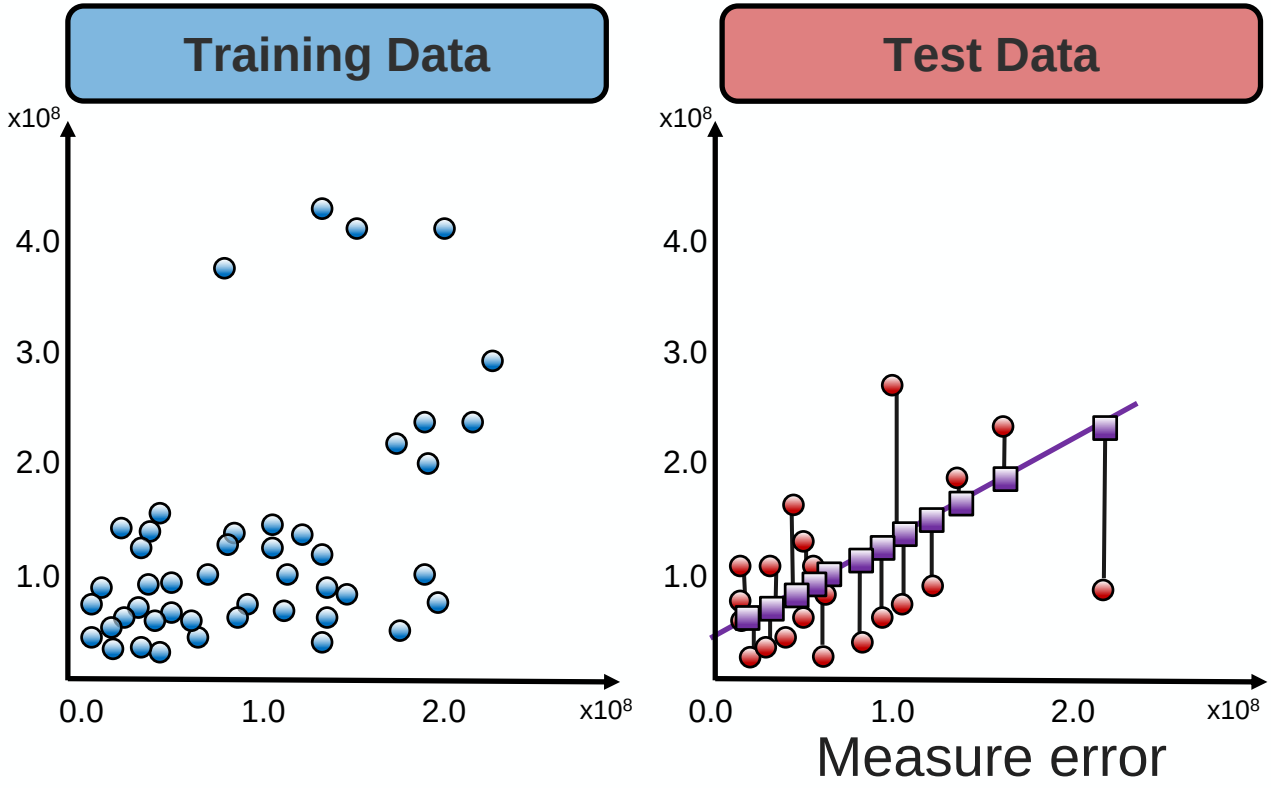
Fit the model

Using Training and Test Data

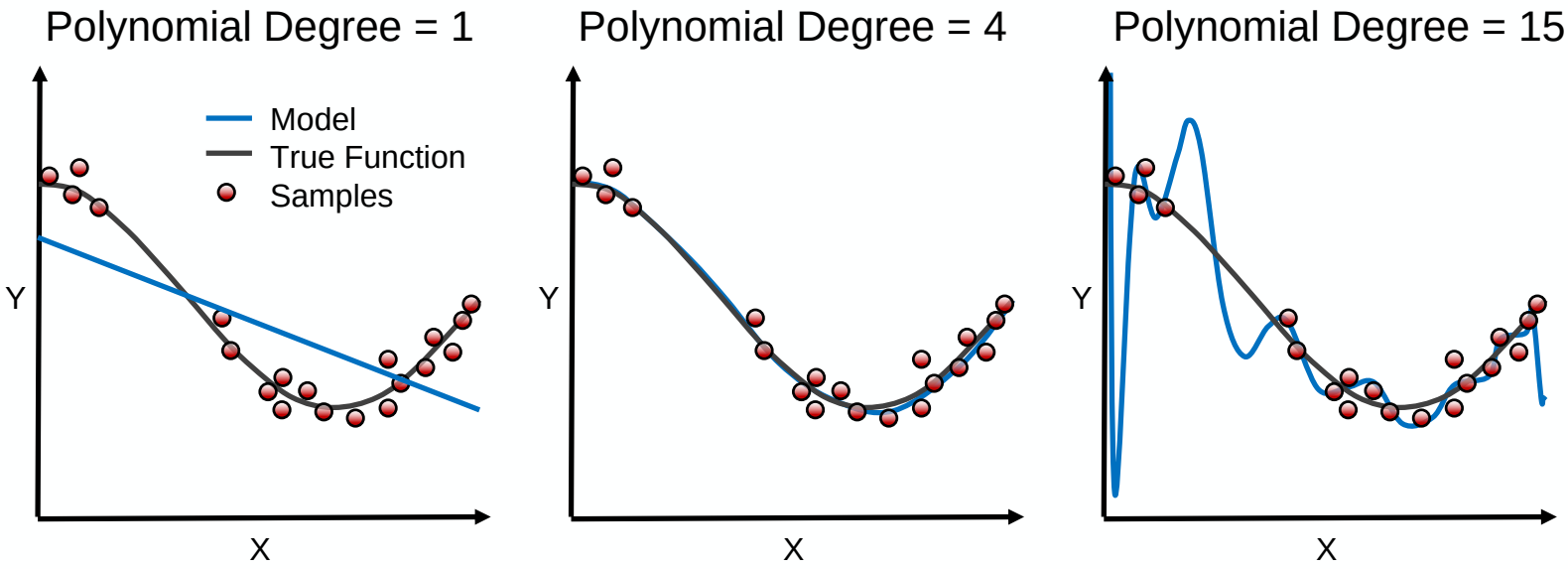


Make predictions

Using Training and Test Data



How Well Does the Model Generalize?

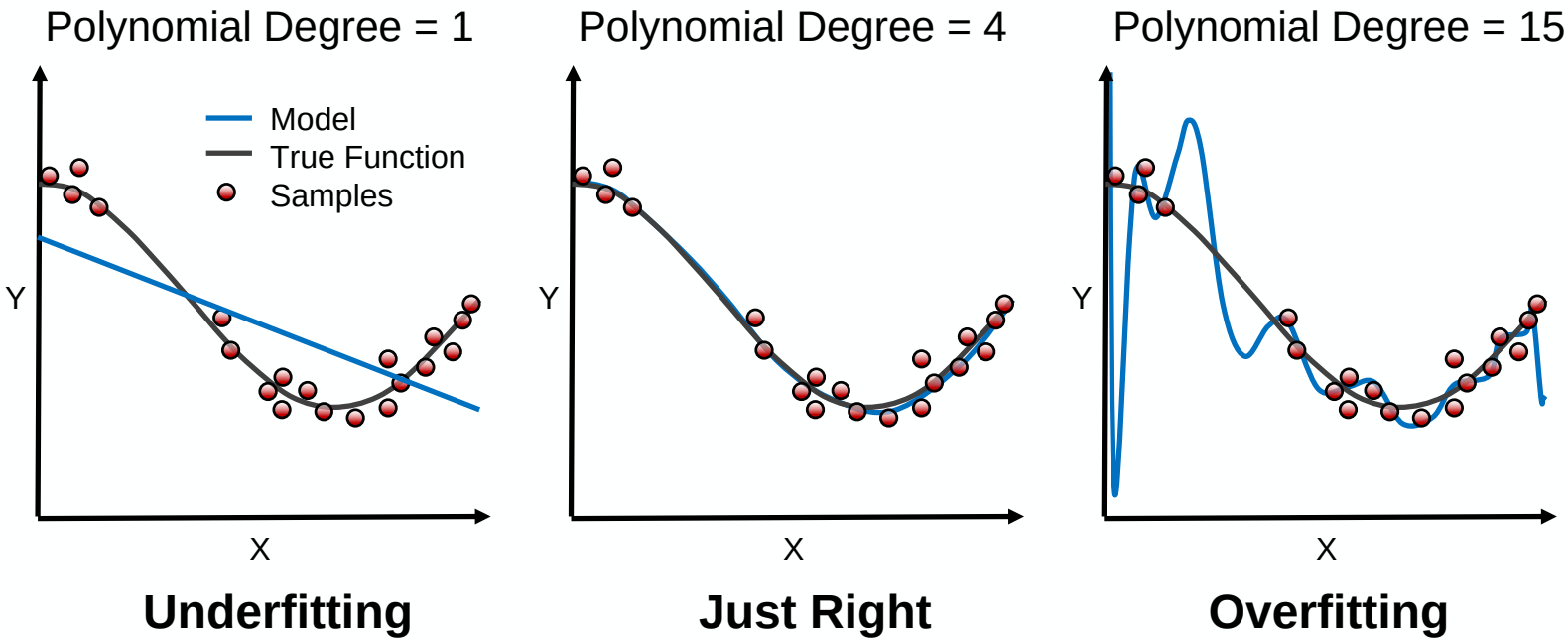


Poor on Training Set
Poor at Predicting

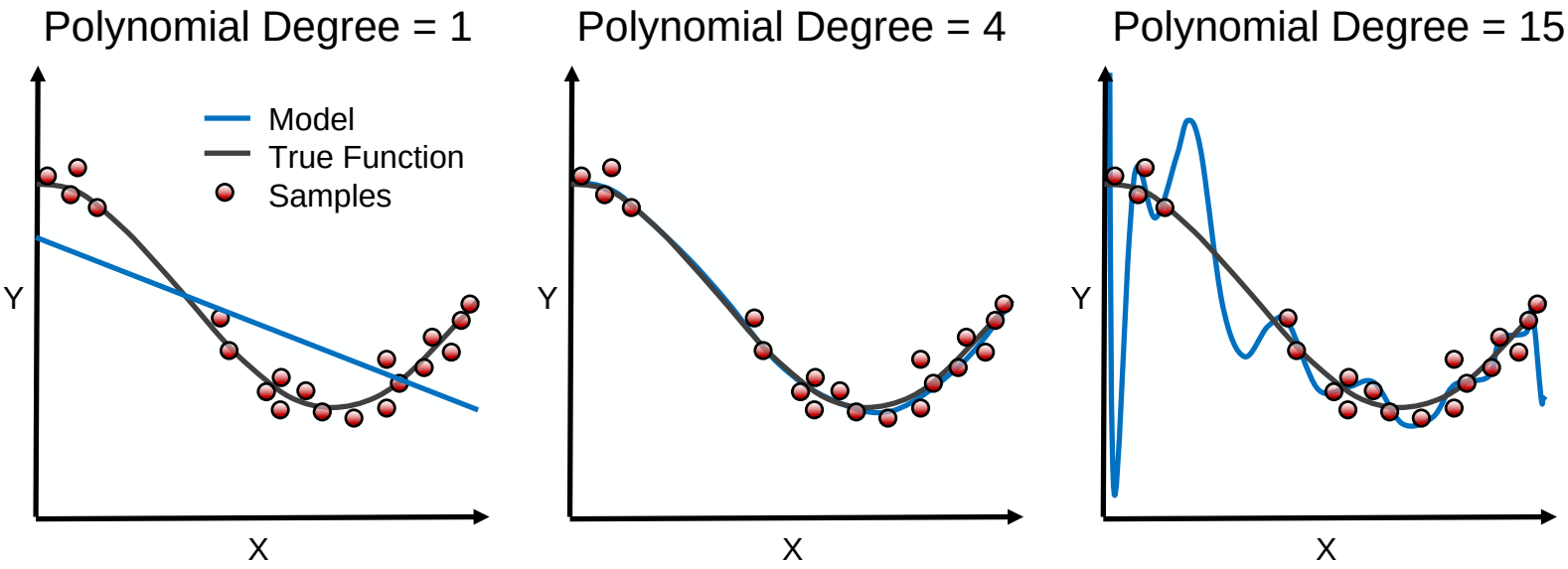
Just Right

Very Good on Training Set
Poor at Predicting

Underfitting vs Overfitting



Bias – Variance Tradeoff



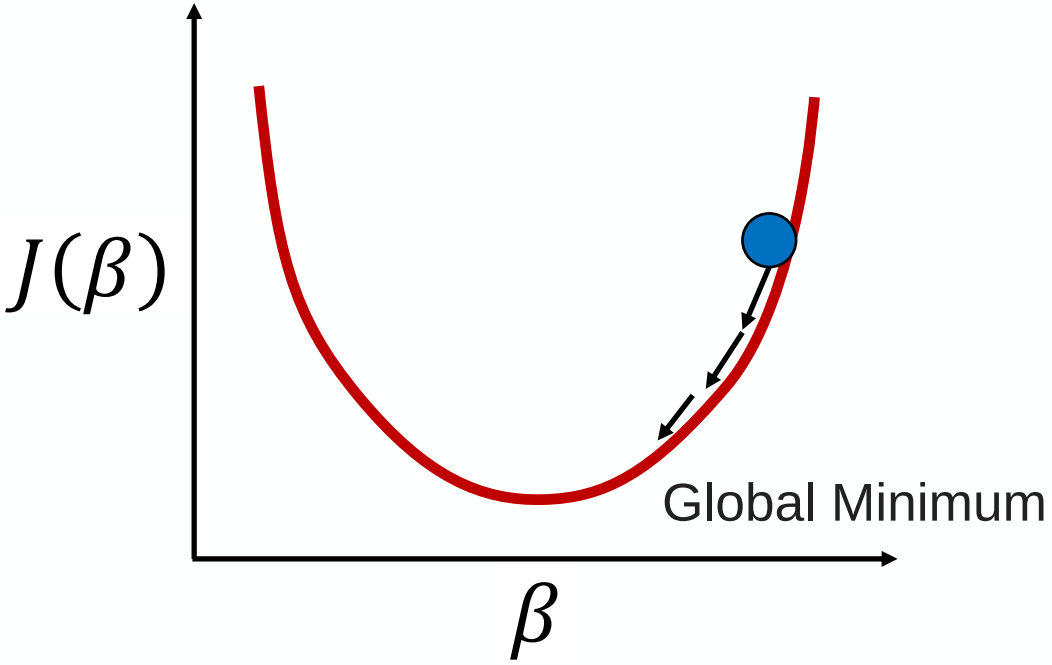
High Bias
Low Variance

Just Right

Low Bias
High Variance

Gradient Descent

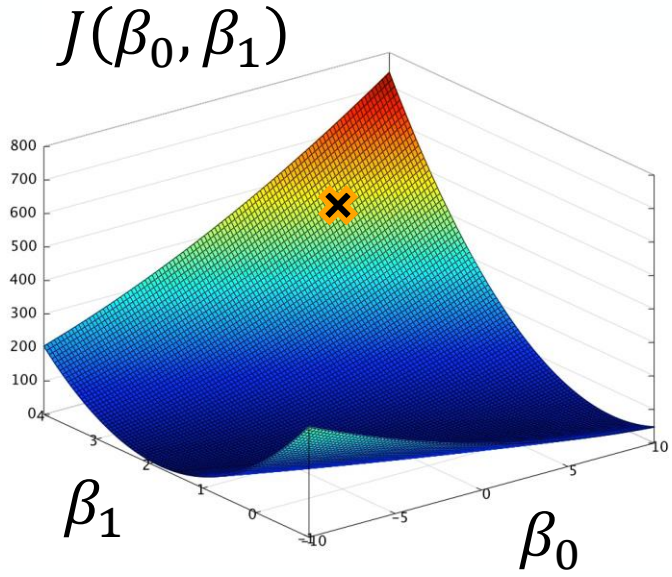
Start with a cost function J(β):



Then gradually move towards the minimum.

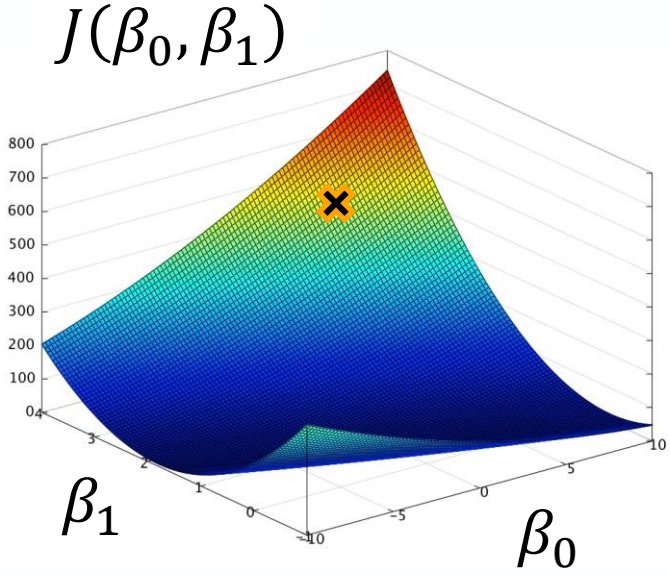
Gradient Descent with Linear Regression

- Now imagine there are two parameters (β_0, β_1)
- This is a more complicated surface on which the minimum must be found
- How can we do this without knowing what $J(\beta_0, \beta_1)$ looks like?



Gradient Descent with Linear Regression

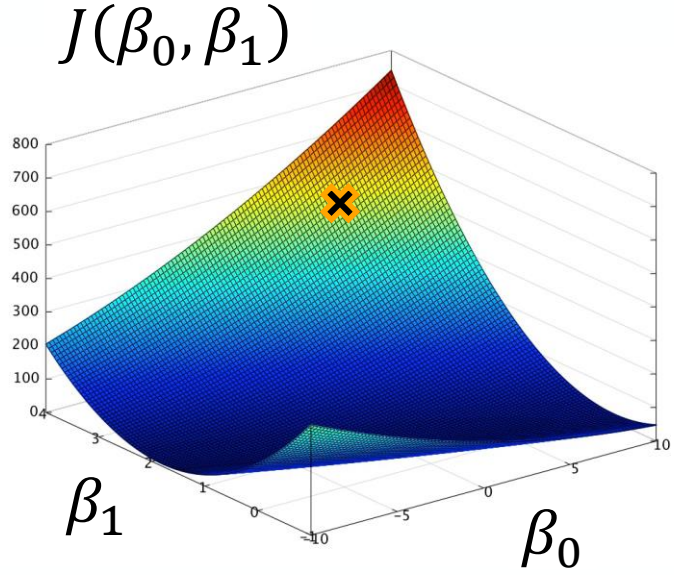
- Compute the gradient, $\nabla J(\beta_0, \beta_1)$, which points in the direction of the biggest increase!
- -$\nabla J(\beta_0, \beta_1)$ (negative gradient) points to the biggest decrease at that point!



Gradient Descent with Linear Regression

- The gradient is the a vector whose coordinates consist of the partial derivatives of the parameters

$$\nabla J(\beta_0, \ldots, \beta_n) = < \frac{\partial J}{\partial \beta_0}, \ \ldots, \frac{\partial J}{\partial \beta_n} >$$

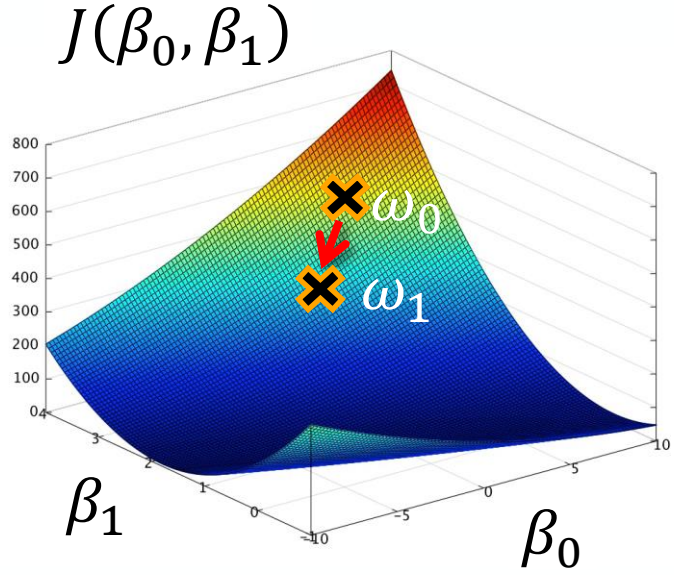


Gradient Descent with Linear Regression

- Then use the gradient (∇) and the cost function to calculate the next point (ω_1) from the current one (ω_0):

$$\omega_1 = \omega_0 - \alpha\nabla\frac{1}{2}\sum_{i=1}^{m}\left(\left(\beta_0 + \beta_1 x_{obs}^{(i)}\right) - y_{obs}^{(i)}\right)^2$$

- The learning rate (α) is a tunable parameter that determines step size

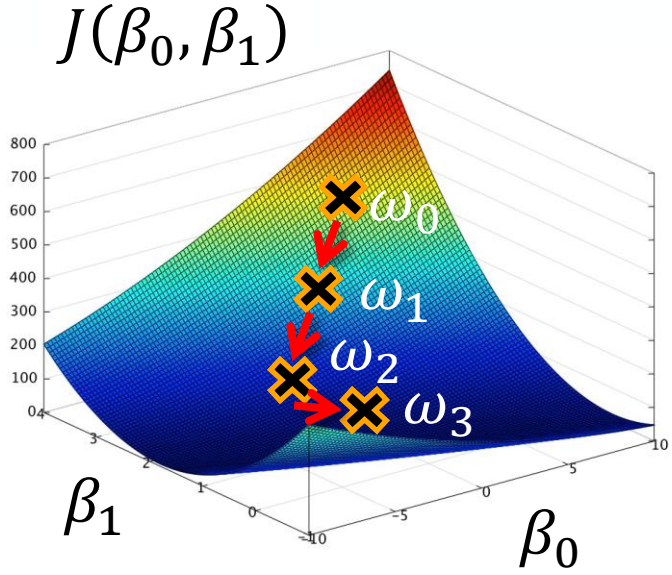


Gradient Descent with Linear Regression

- Each point can be iteratively calculated from the previous one

$$\omega_2 = \omega_1 - \alpha\nabla\frac{1}{2}\sum_{i=1}^{m}\left(\left(\beta_0 + \beta_1 x_{obs}^{(i)}\right) - y_{obs}^{(i)}\right)^2$$

$$\omega_3 = \omega_2 - \alpha\nabla\frac{1}{2}\sum_{i=1}^{m}\left(\left(\beta_0 + \beta_1 x_{obs}^{(i)}\right) - y_{obs}^{(i)}\right)^2$$

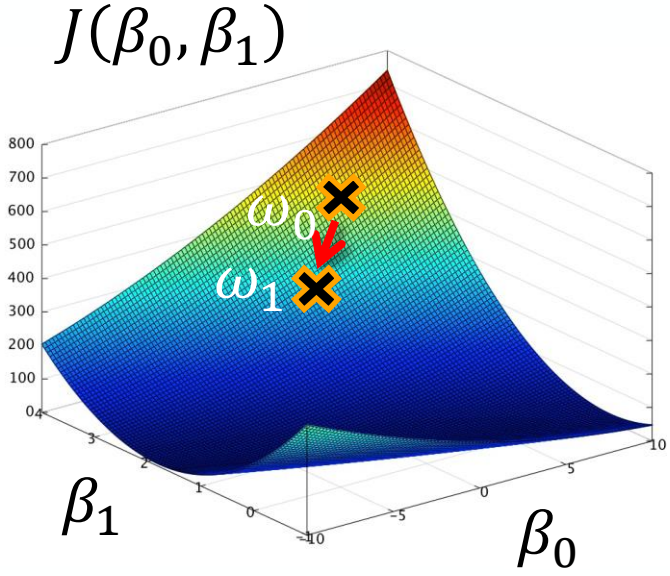


Stochastic Gradient Descent

- Use a single data point to determine the gradient and cost function instead of all the data

$$\omega_1 = \omega_0 - \alpha\nabla\frac{1}{2}\sum_{i=1}^{m}\left(\left(\beta_0 + \beta_1 x_{obs}^{(i)}\right) - y_{obs}^{(i)}\right)^2$$

$$\omega_1 = \omega_0 - \alpha\nabla\frac{1}{2}\left(\left(\beta_0 + \beta_1 x_{obs}^{(0)}\right) - y_{obs}^{(0)}\right)^2$$



Stochastic Gradient Descent

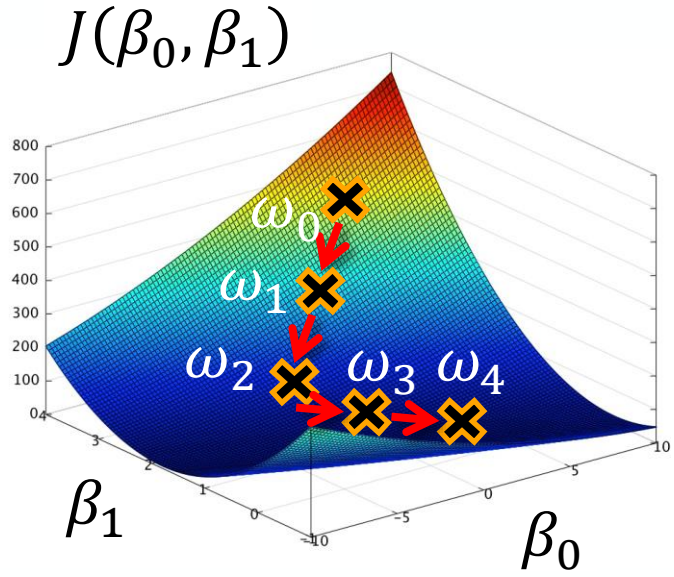
- Use a single data point to determine the gradient and cost function instead of all the data

$$\omega_1 = \omega_0 - \alpha\nabla\frac{1}{2}\left(\left(\beta_0 + \beta_1 x_{obs}^{(0)}\right) - y_{obs}^{(0)}\right)^2$$

…

$$\omega_4 = \omega_3 - \alpha\nabla\frac{1}{2}\left(\left(\beta_0 + \beta_1 x_{obs}^{(3)}\right) - y_{obs}^{(3)}\right)^2$$

- Path is less direct due to noise in single data point—"stochastic"



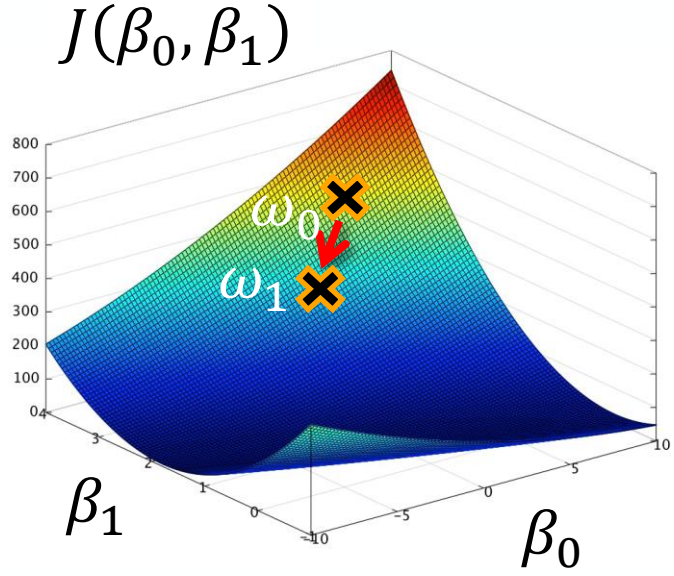
Mini Batch Gradient Descent

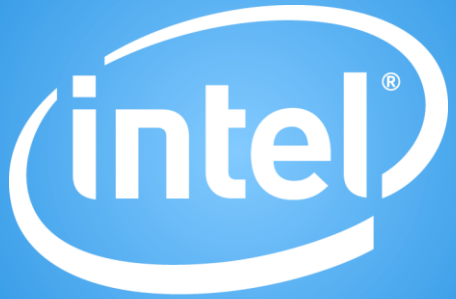
- Perform an update for every n training examples

$$\omega_1 = \omega_0 - \alpha\nabla\frac{1}{2}\sum_{i=1}^{n}\left(\left(\beta_0 + \beta_1 x_{obs}^{(i)}\right) - y_{obs}^{(i)}\right)^2$$

Best of both worlds:

- Reduced memory relative to "vanilla" gradient descent
- Less noisy than stochastic gradient descent




intel
Software

Introduction to Neural Nets

Legal Notices and Disclaimers

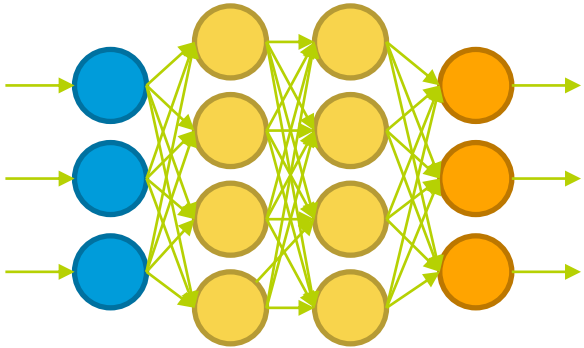
This presentation is for informational purposes only. INTEL MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at intel.com.

This sample source code is released under the Intel Sample Source Code License Agreement.

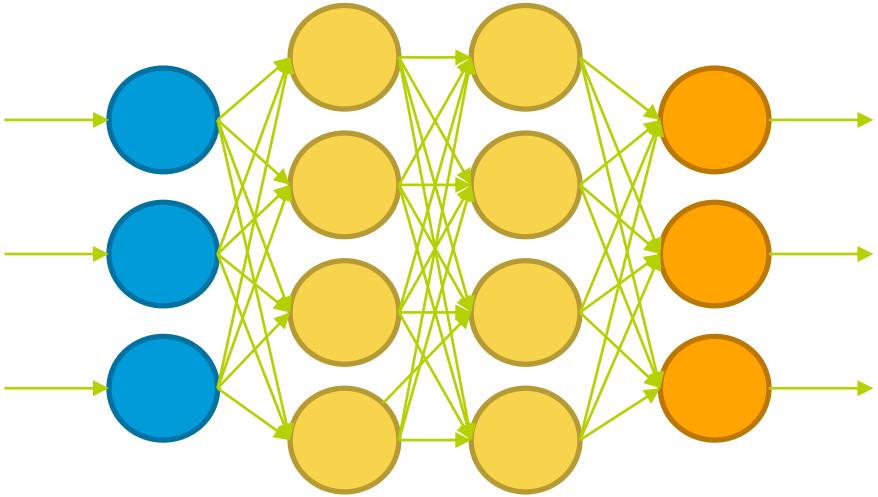
Motivation for Neural Nets

- Use biology as inspiration for mathematical model
- Get signals from previous neurons
- Generate signals (or not) according to inputs
- Pass signals on to next neurons
- By layering many neurons, can create complex model



Neural Net Structure

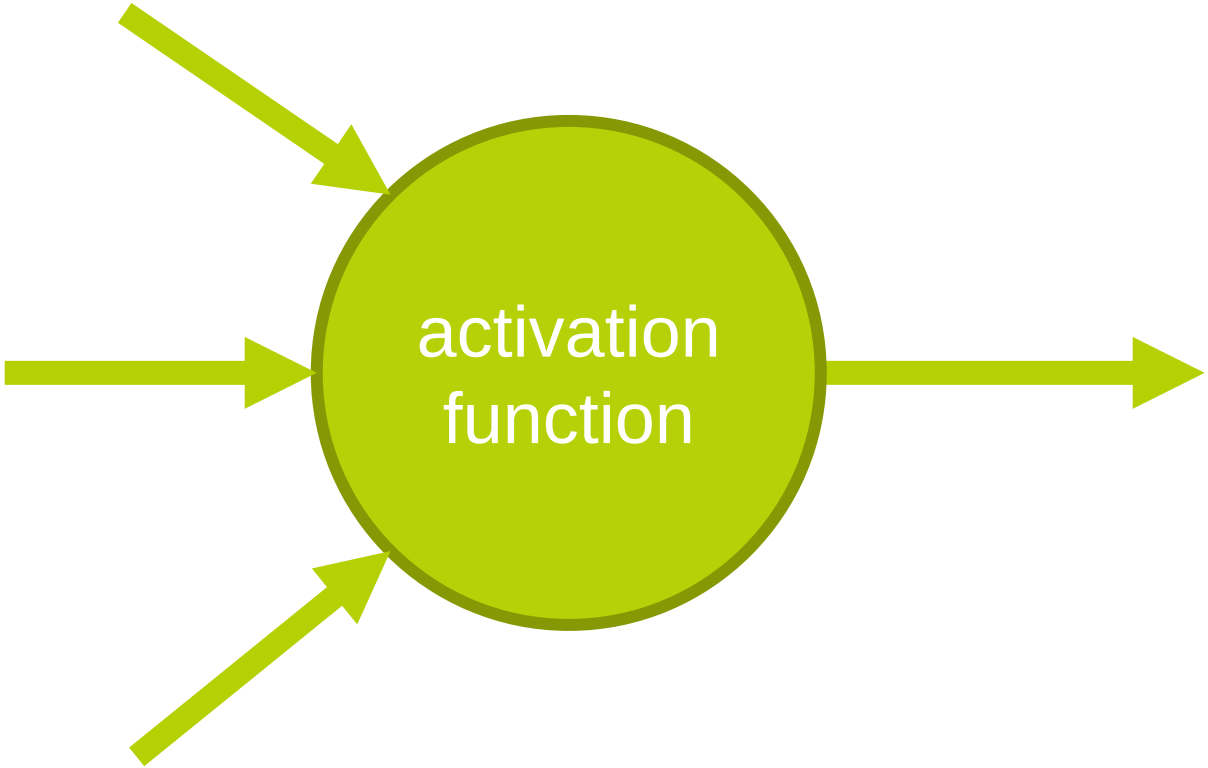
Input
(Feature Vector)



Output
(Label)

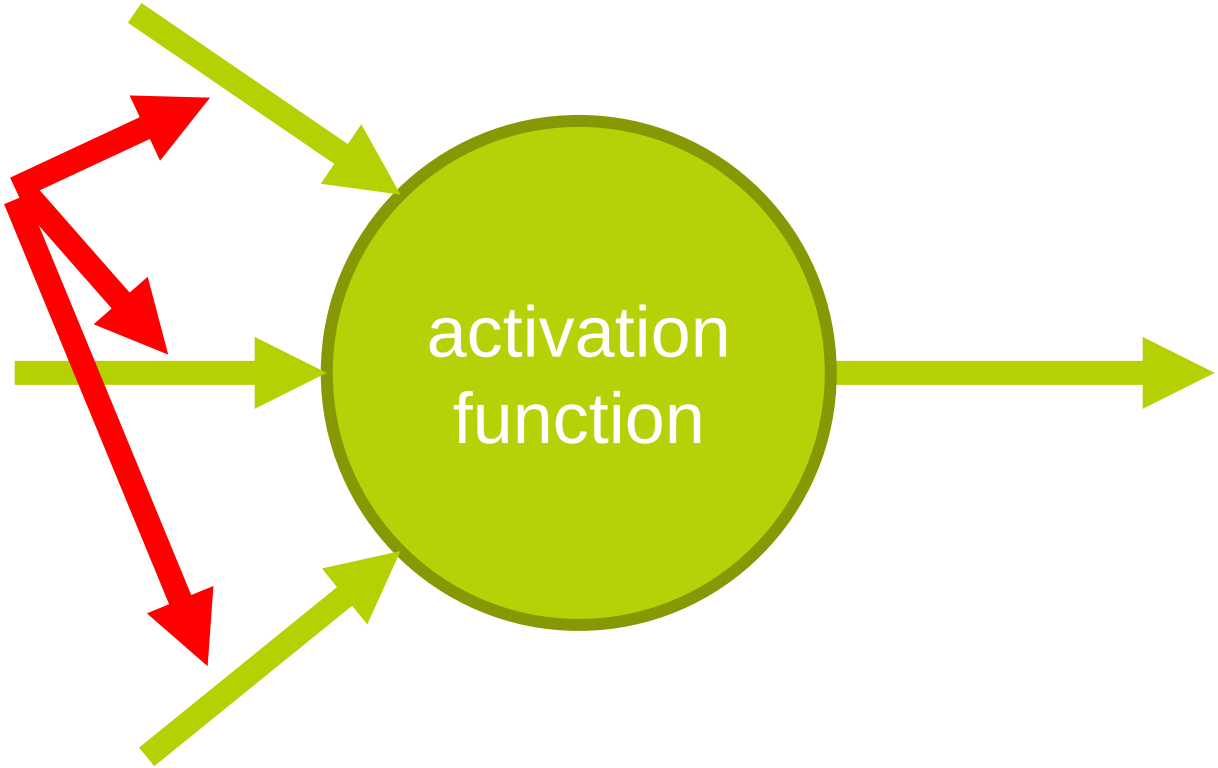
- Can think of it as a complicated computation engine
- We will ”train it” using our training data
- Then (hopefully) it will give good answers on new data

Basic Neuron Visualization

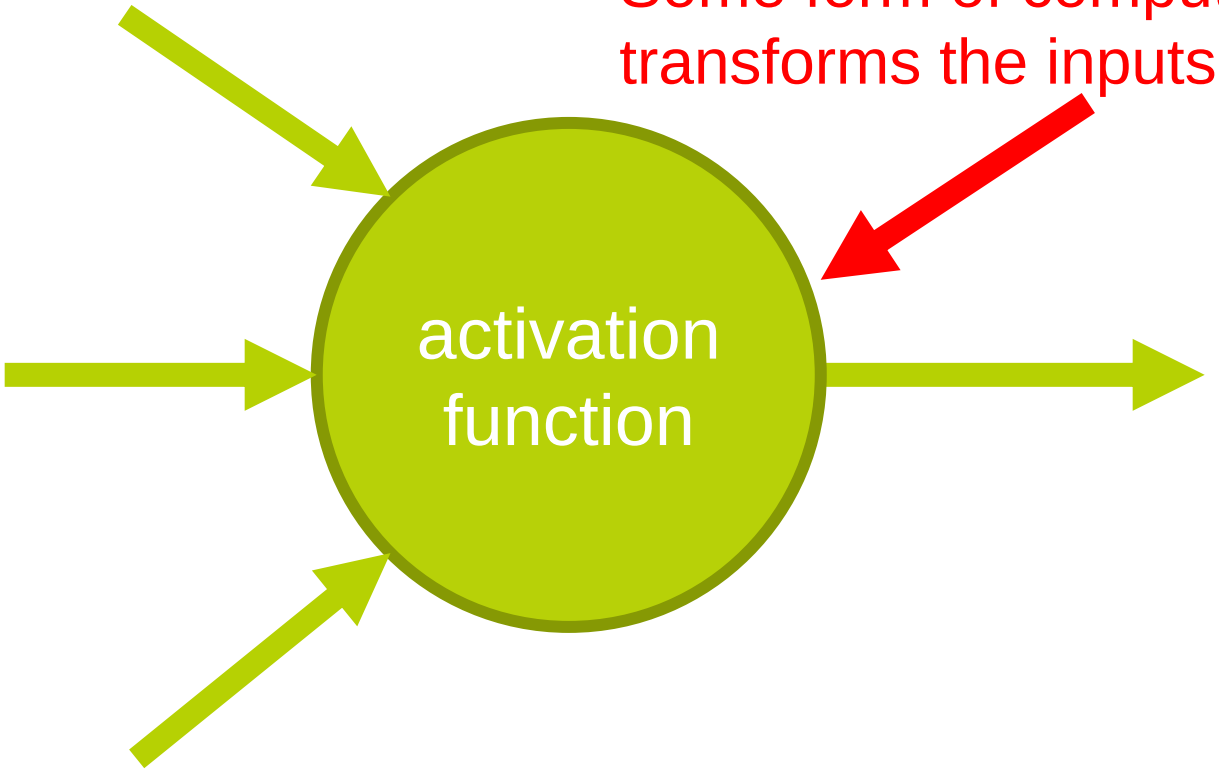


Basic Neuron Visualization

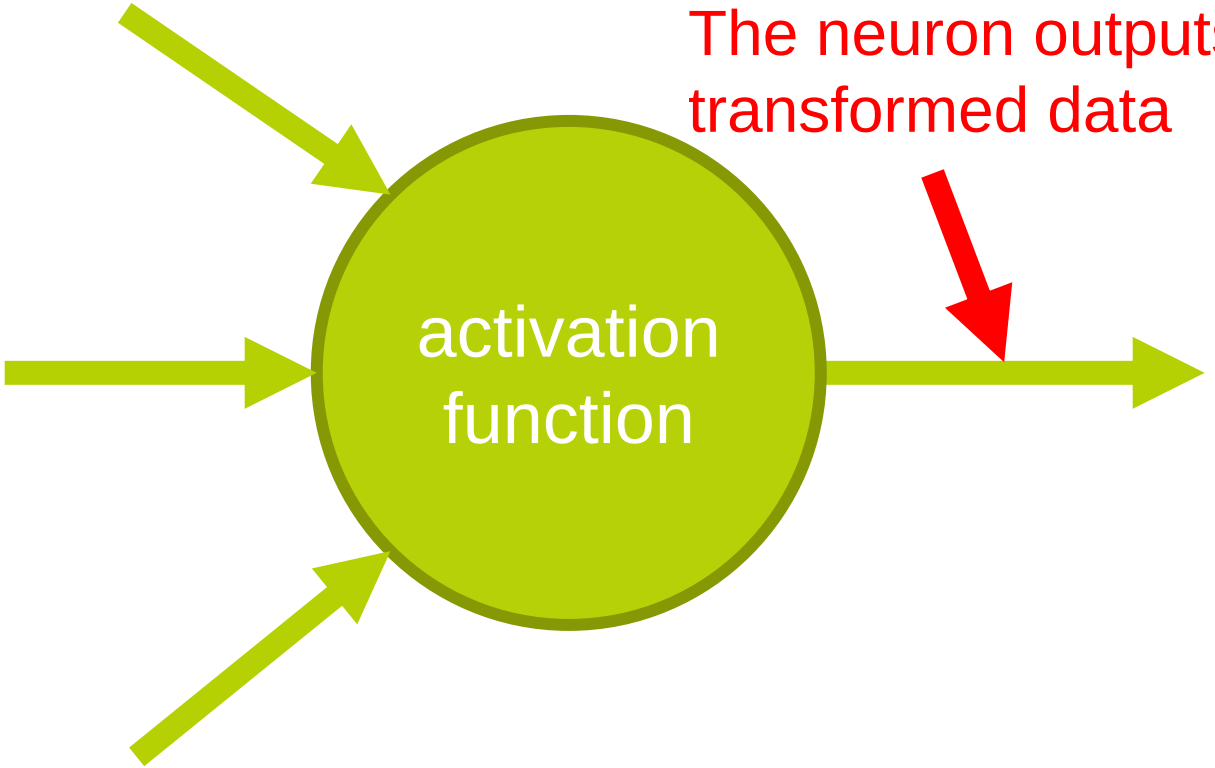
Data from previous layer



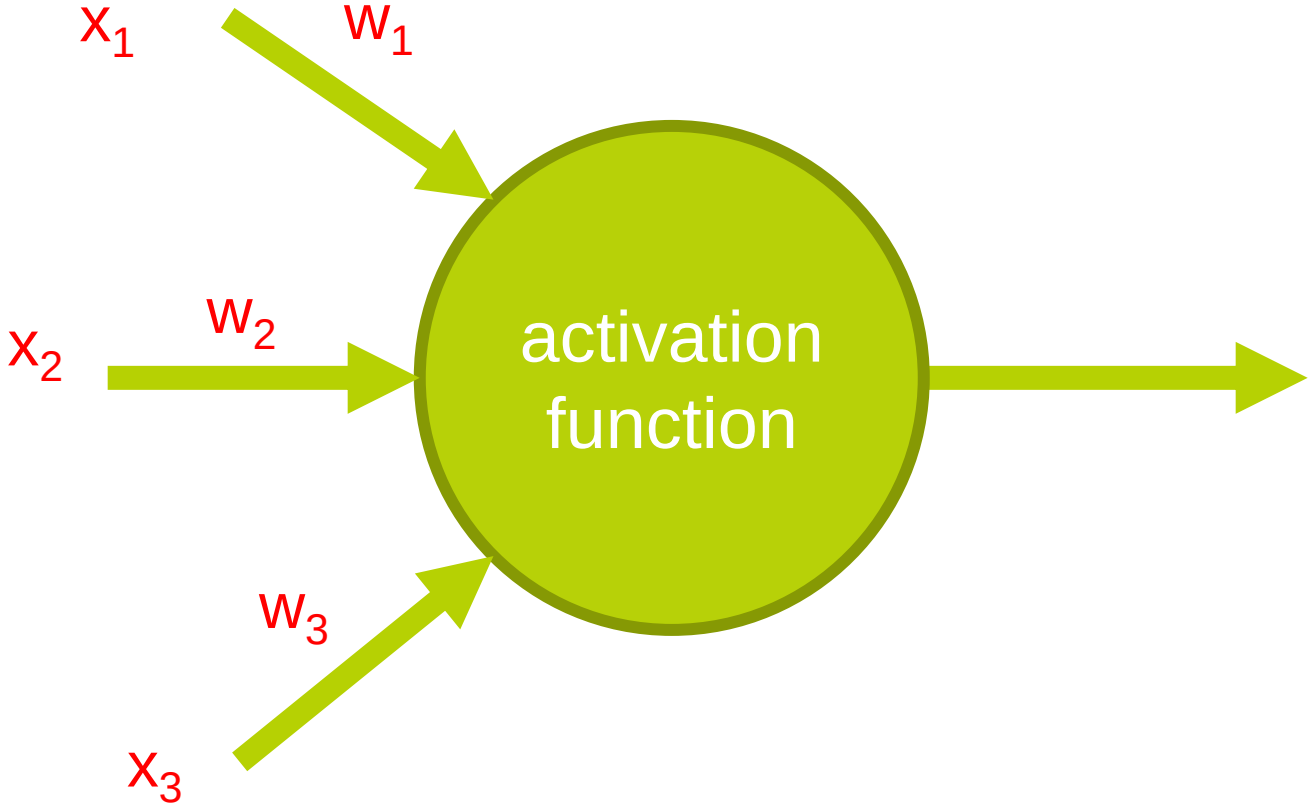
Basic Neuron Visualization



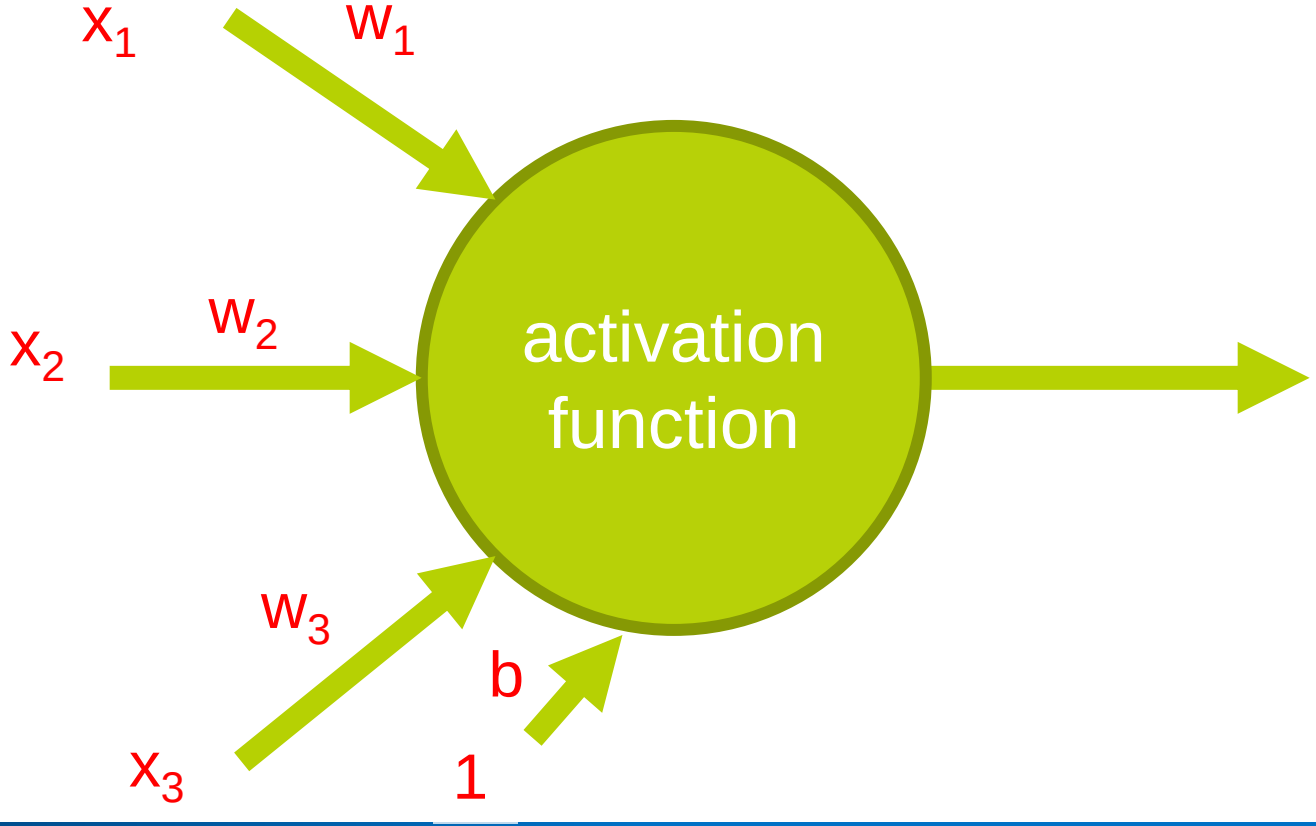
Basic Neuron Visualization



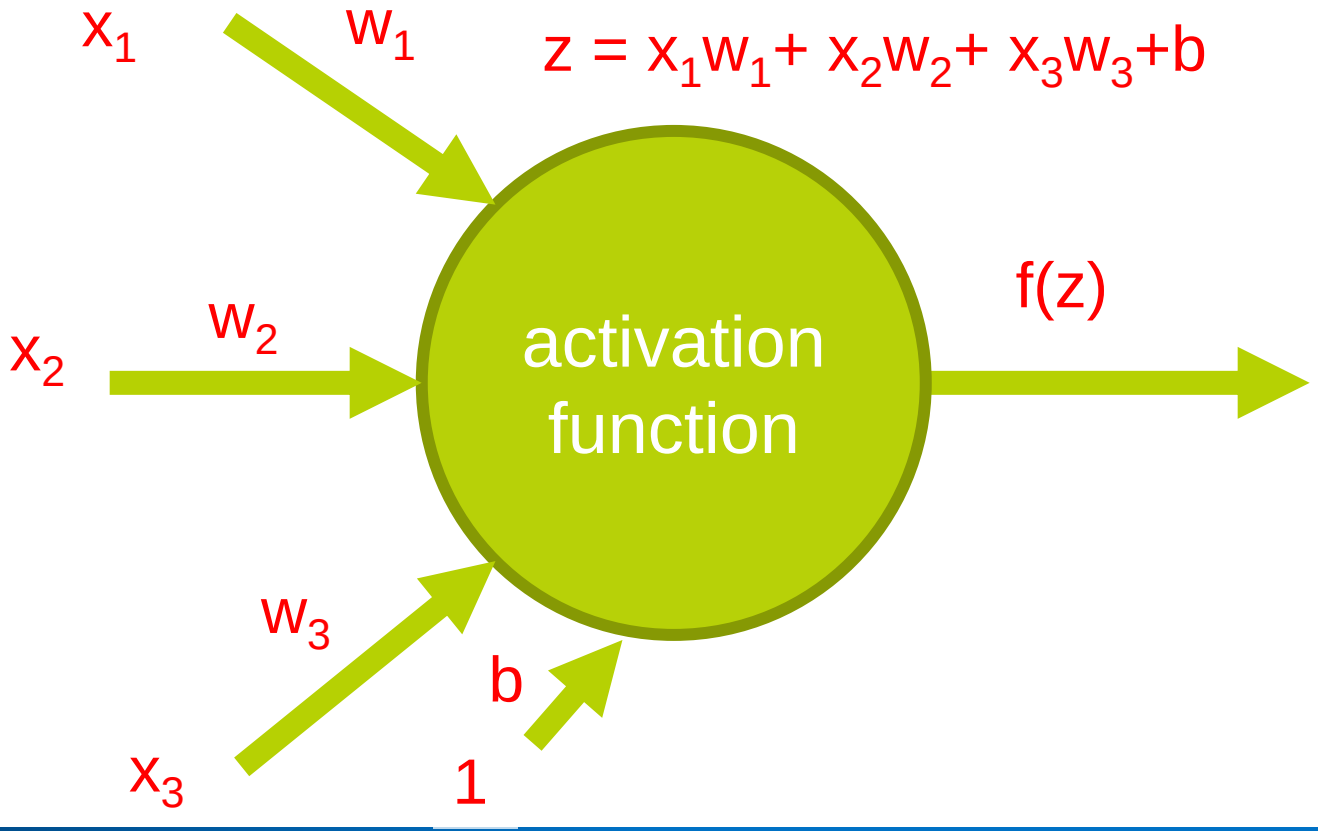
Basic Neuron Visualization



Basic Neuron Visualization



Basic Neuron Visualization



In Vector Notation

z = “net input”

b = “bias term”

f = activation function

a = output to next layer

$$z = b + \sum_{i=1}^{m} x_i w_i$$

$$z = b + x^T w$$

$$a = f(z)$$

Relation to Logistic Regression

When we choose: $f(z) = \frac{1}{1+e^{-z}}$

$$z = b + \sum_{i=1}^{m} x_i w_i = x_1 w_1 + x_2 w_2 + \cdots + x_m w_m + b$$

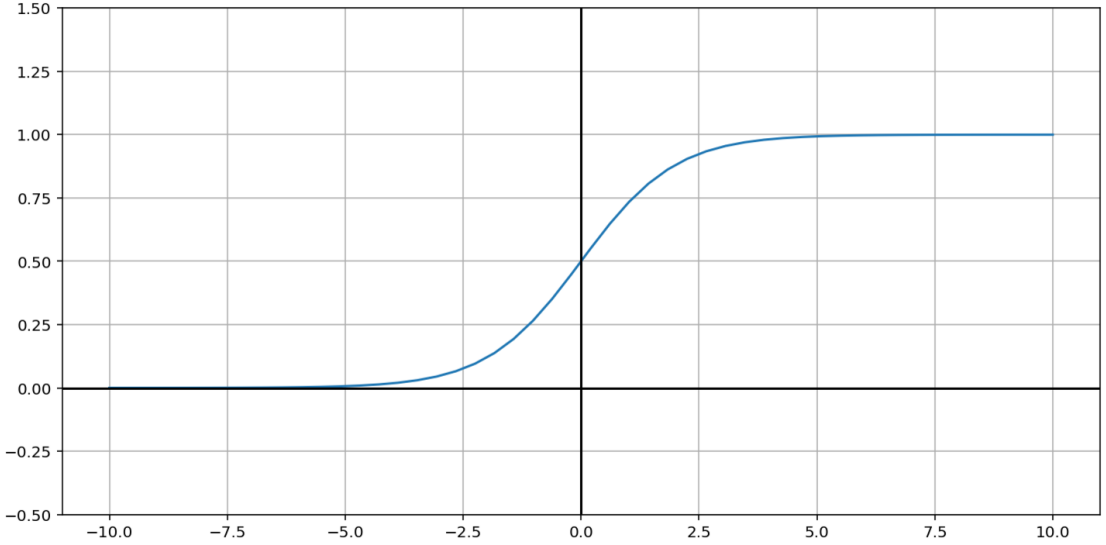
Then a neuron is simply a ”unit” of logistic regression!

weights ⇔ coefficients inputs ⇔ variables

bias term ⇔ constant term

Relation to Logistic Regression

This is called the “sigmoid” function: $\sigma(z) = \frac{1}{1+e^{-z}}$



Nice Property of Sigmoid Function

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

Quotient rule

$$\frac{d}{dx}\cdot\frac{f(x)}{g(x)} = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2}$$

$$\sigma'(z) = \frac{0-(-e^{-z})}{(1+e^{-z})^2} = \frac{e^{-z}}{(1+e^{-z})^2}$$

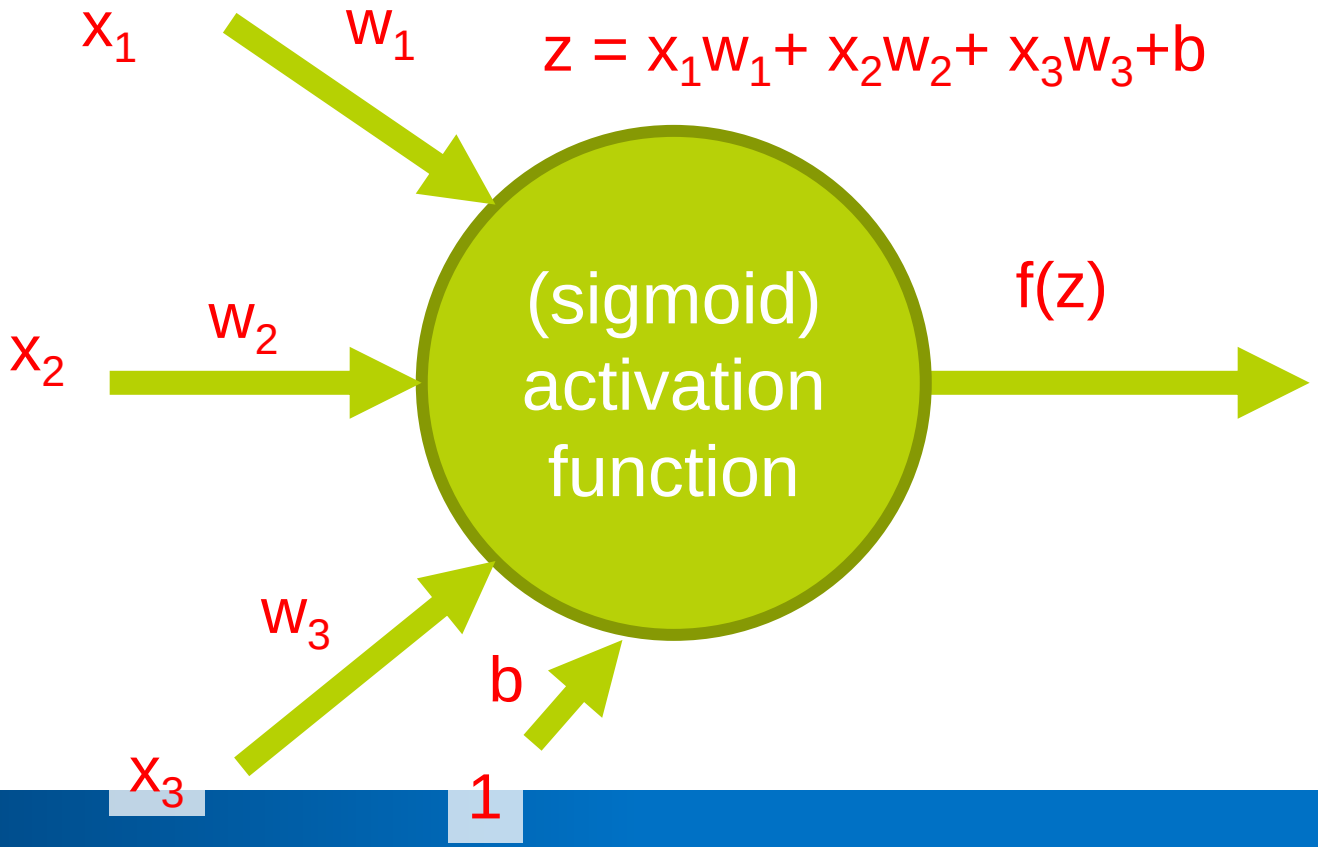
$$= \frac{1+e^{-z}-1}{(1+e^{-z})^2} = \frac{1+e^{-z}}{(1+e^{-z})^2} - \frac{1}{(1+e^{-z})^2}$$

$$= \frac{1}{1+e^{-z}} - \frac{1}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right)$$

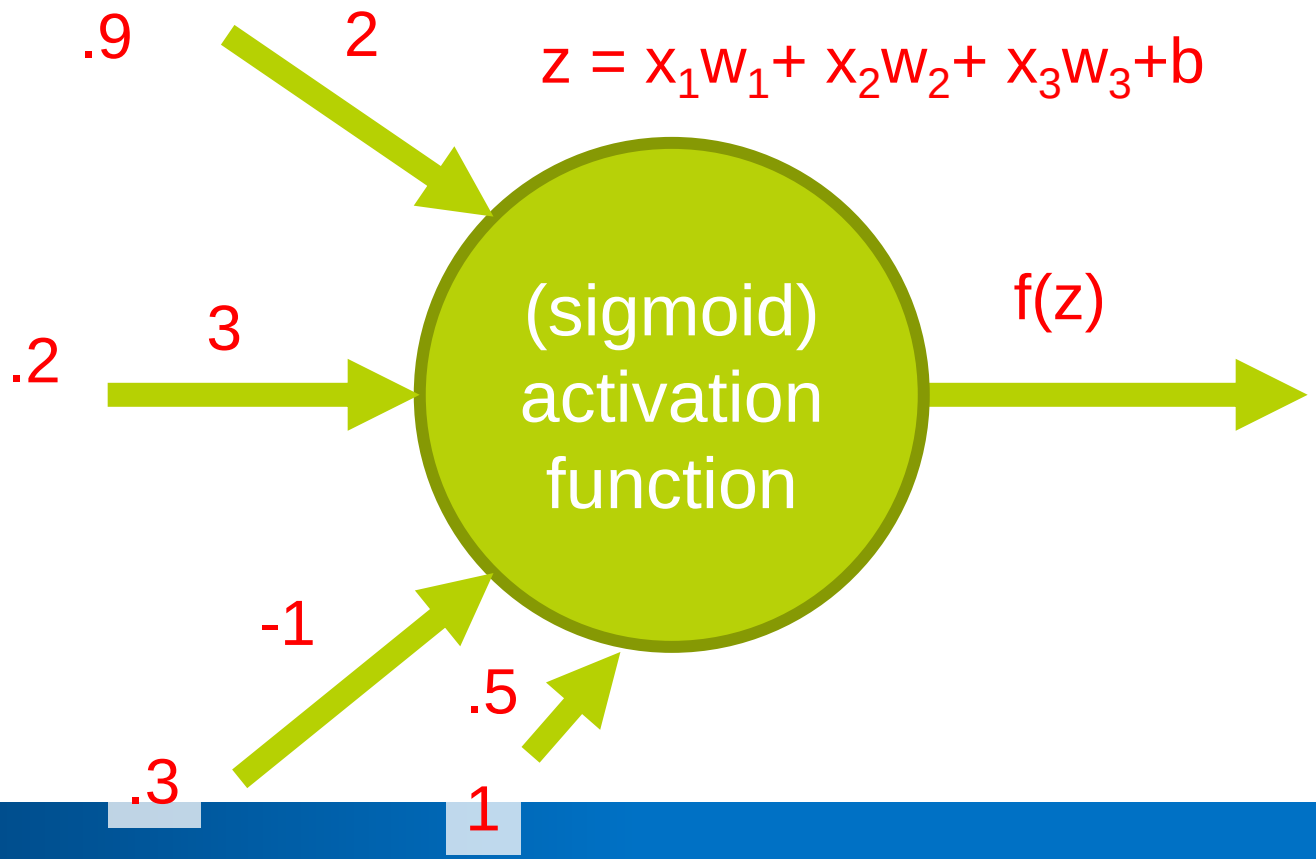
$$\sigma'(z) = \sigma(z)(1-\sigma(z))$$

This will be helpful!

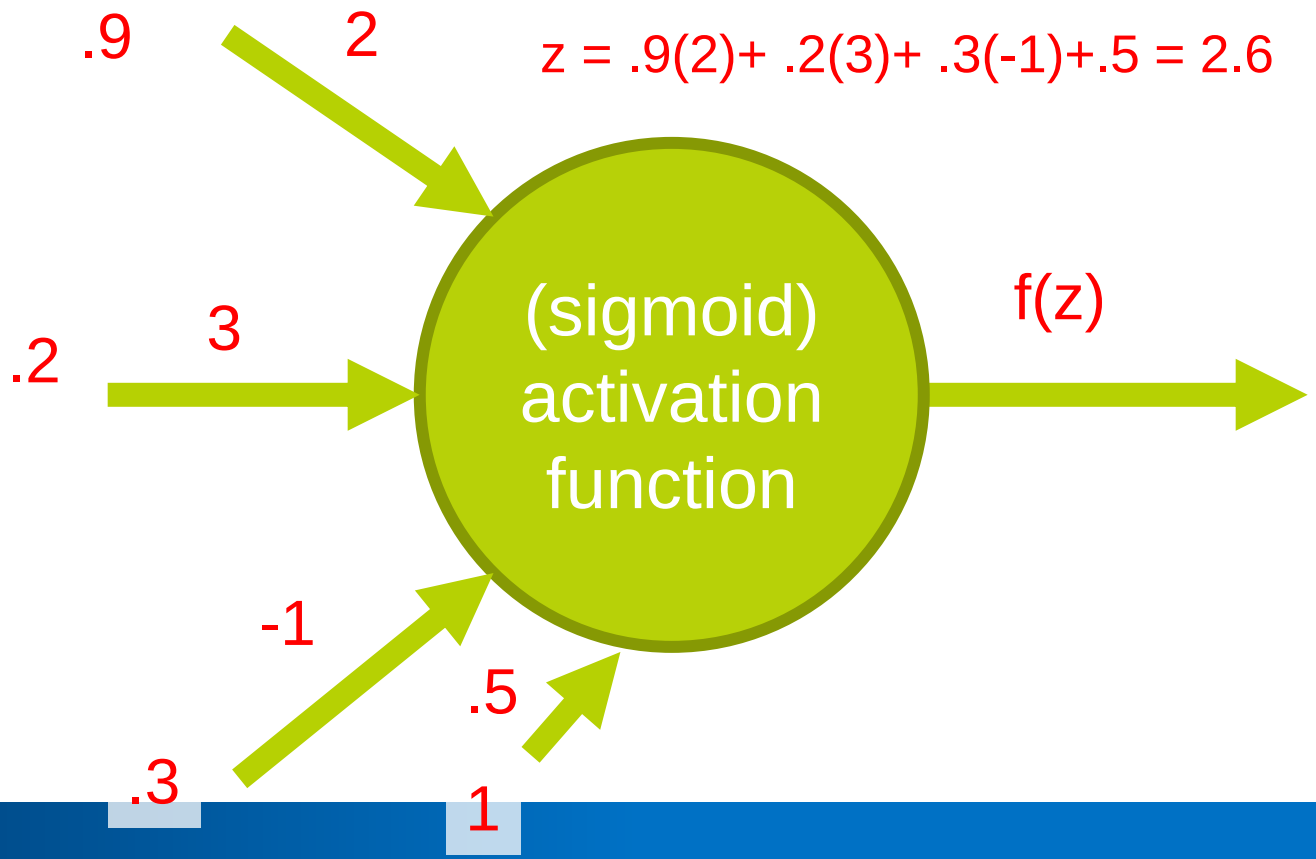
Example Neuron Computation



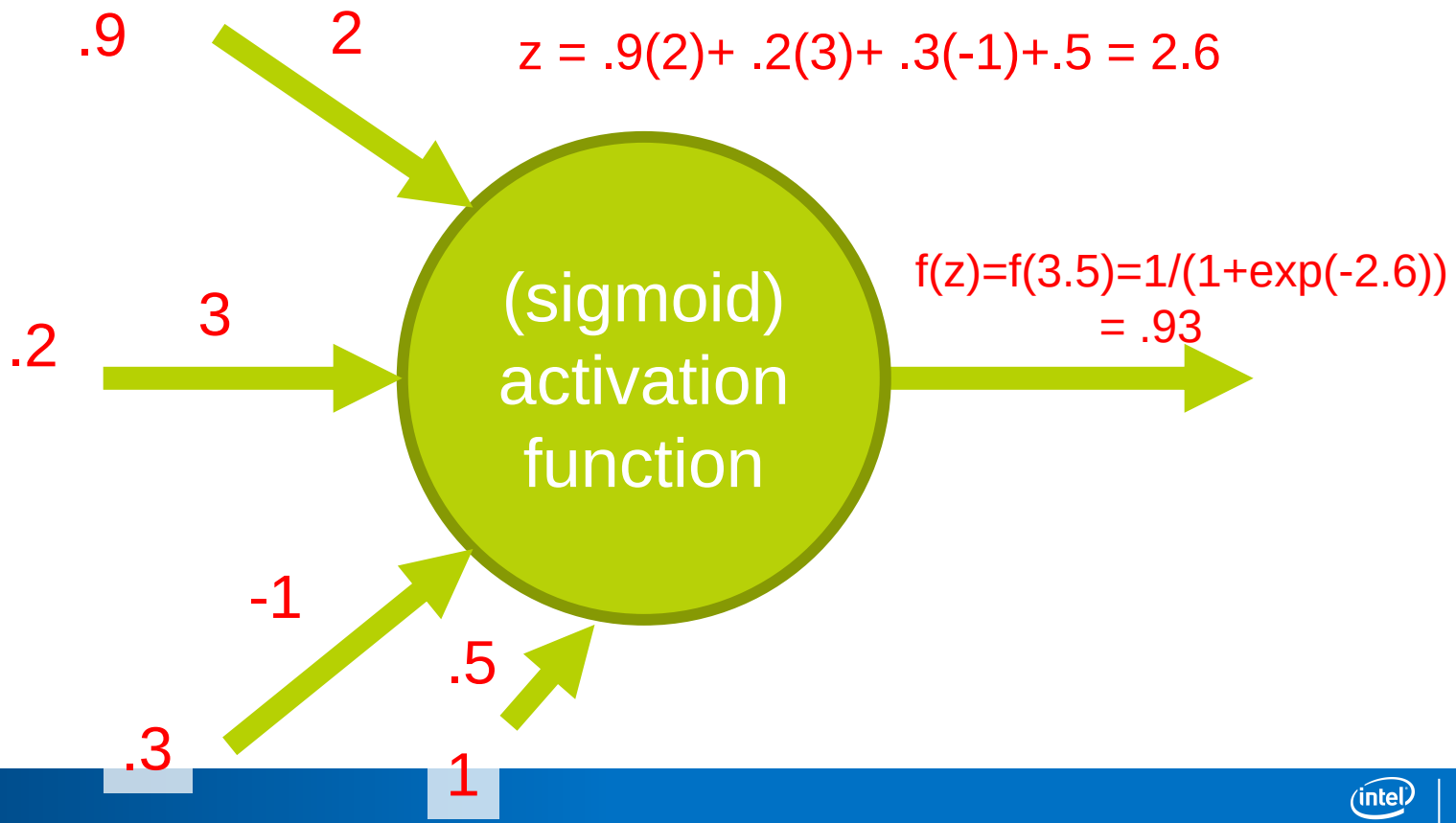
Example Neuron Computation



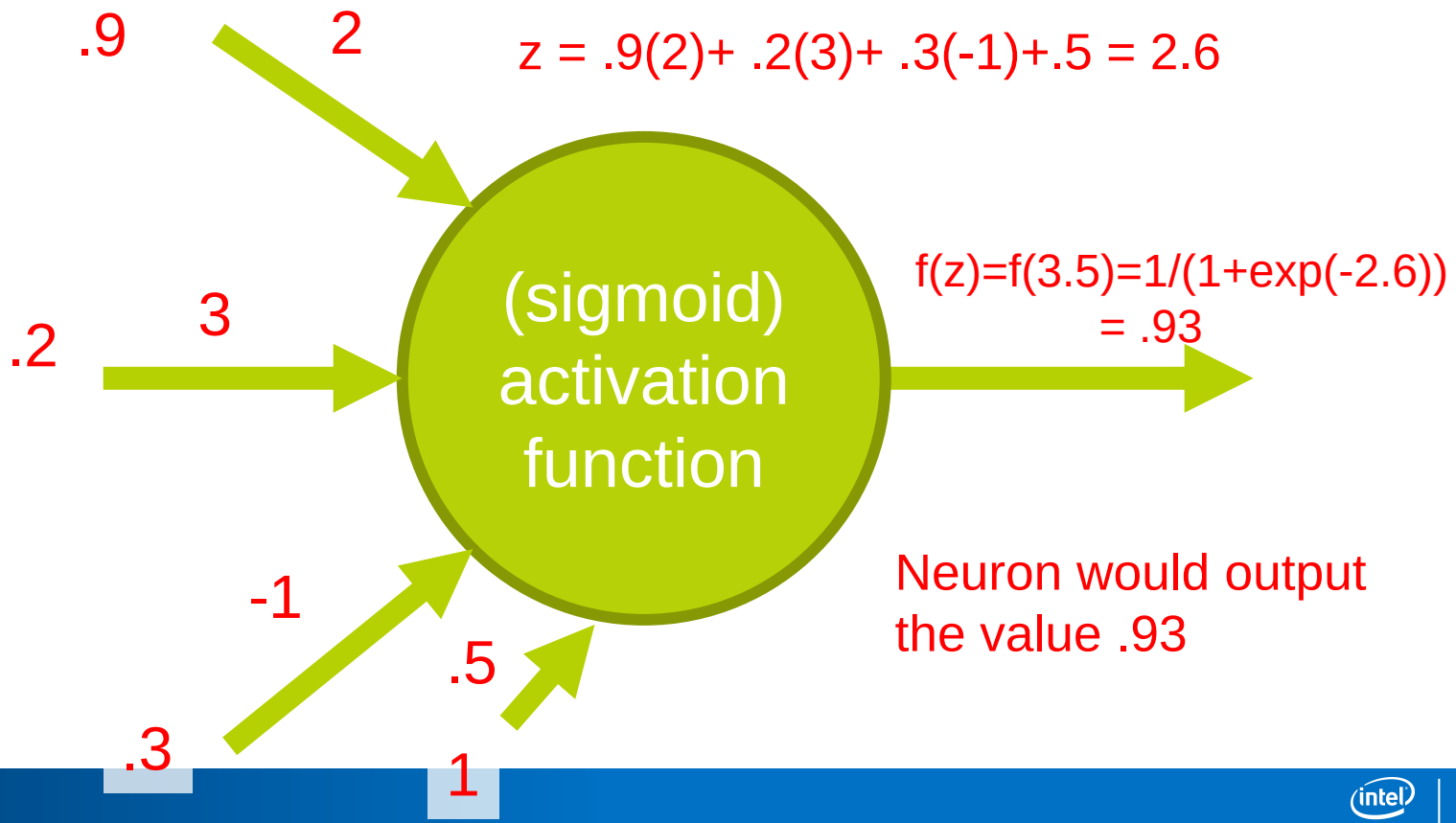
Example Neuron Computation



Example Neuron Computation

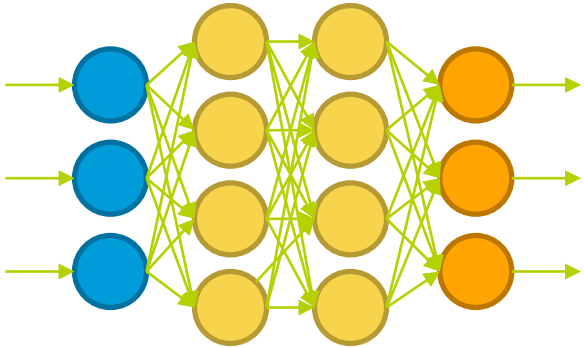


Example Neuron Computation

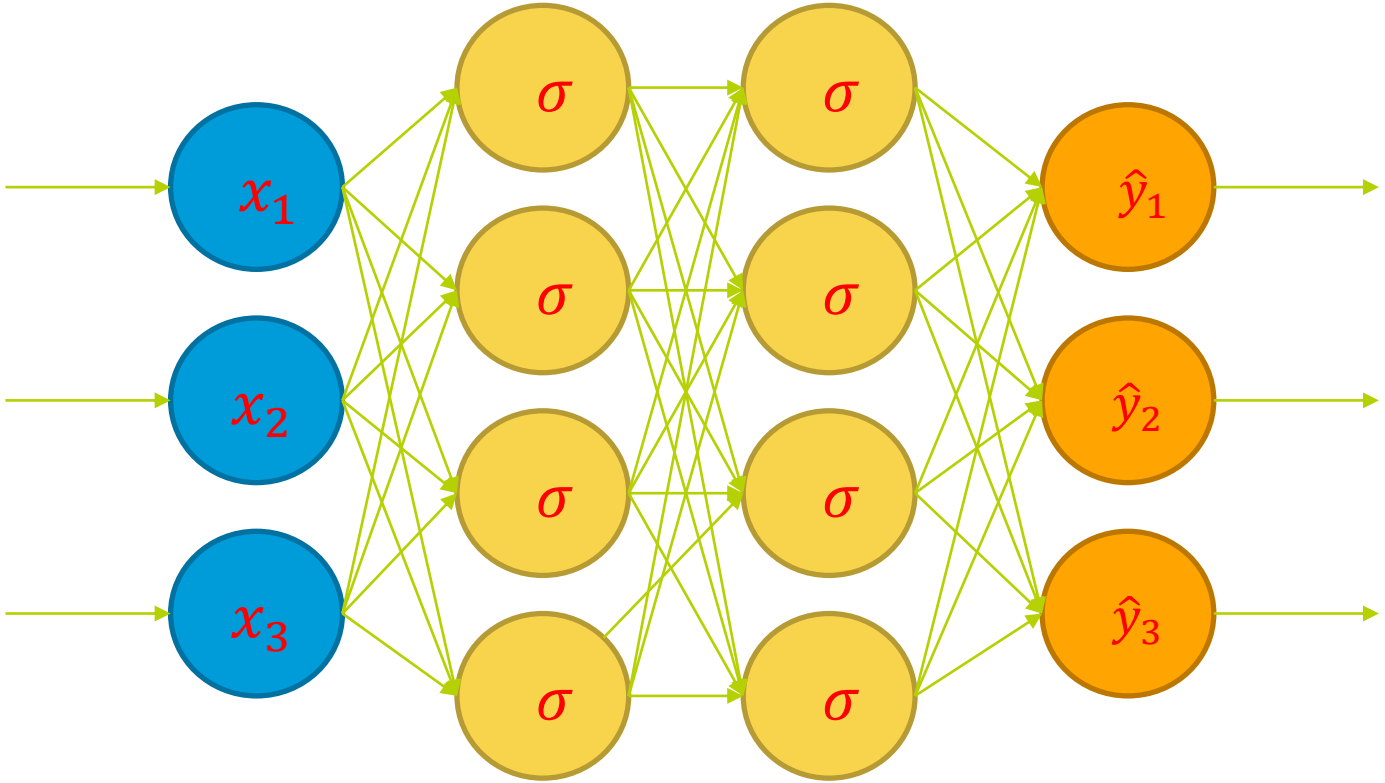


Why Neural Nets?

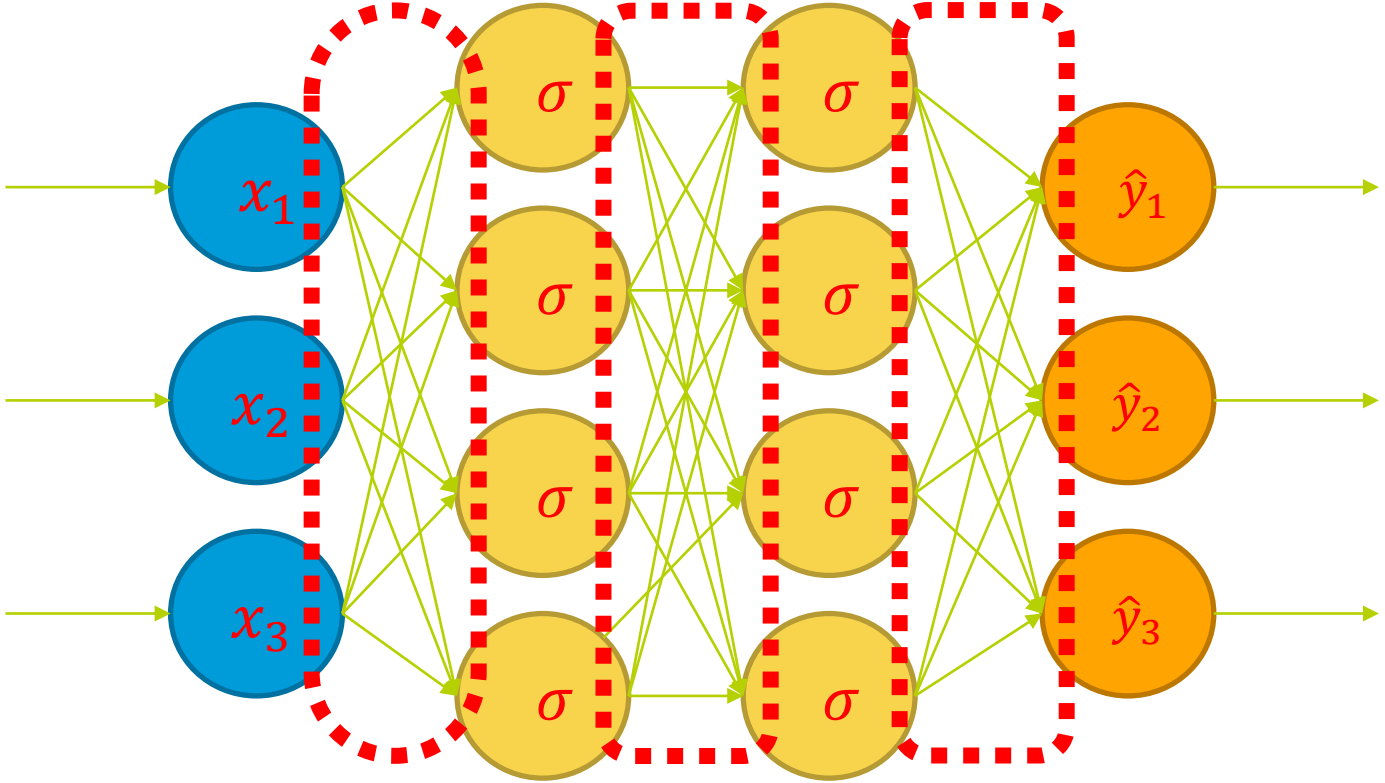
- Why not just use a single neuron? Why do we need a larger network?
- A single neuron (like logistic regression) only permits a linear decision boundary.
- Most real-world problems are considerably more complicated!



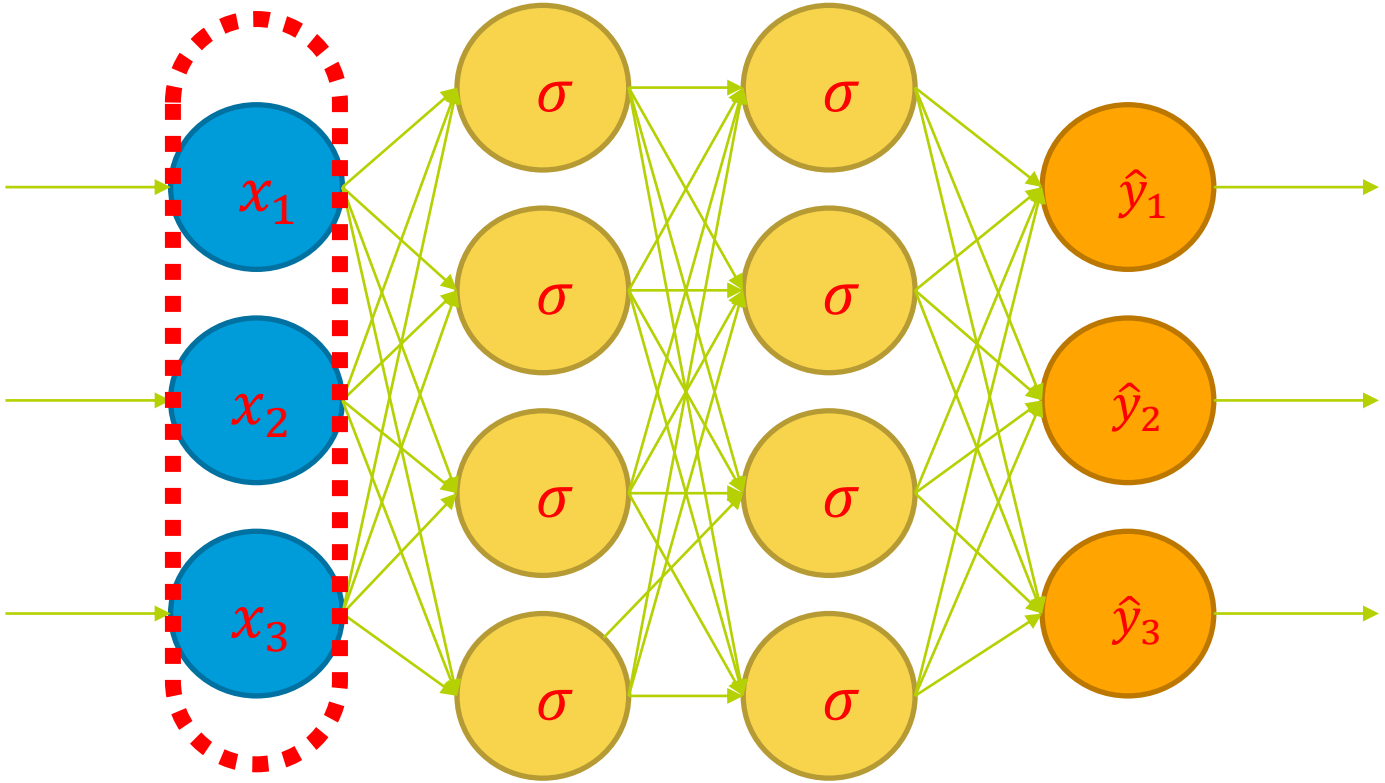
Feedforward Neural Network



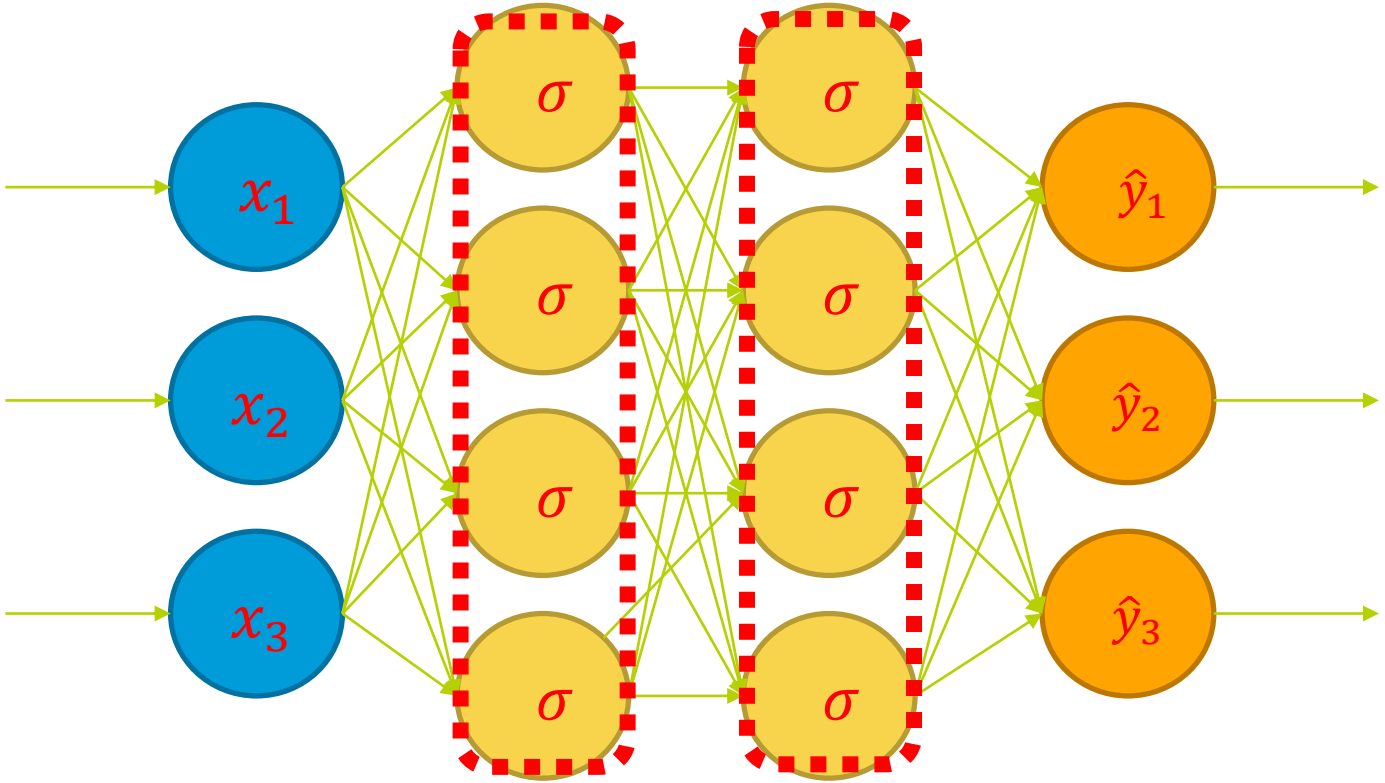
Weights



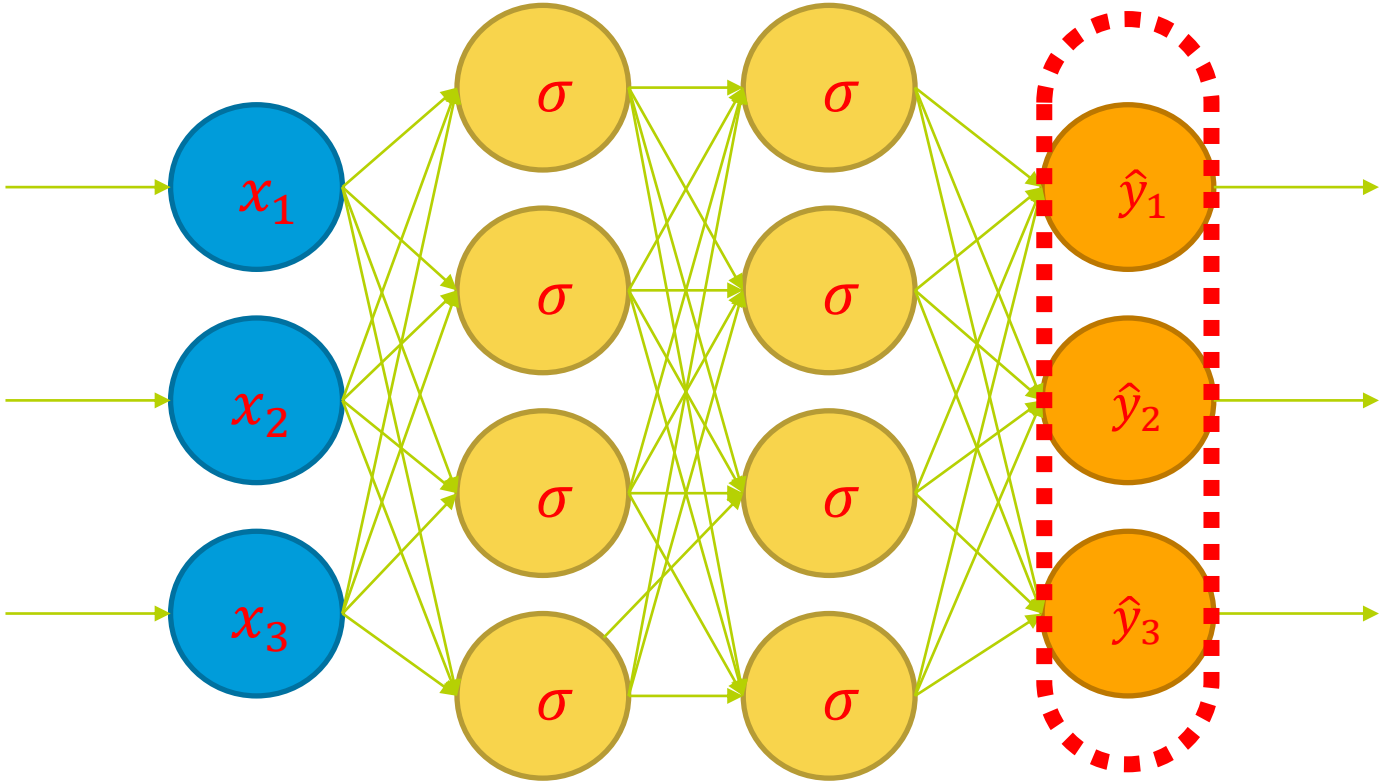
Input Layer



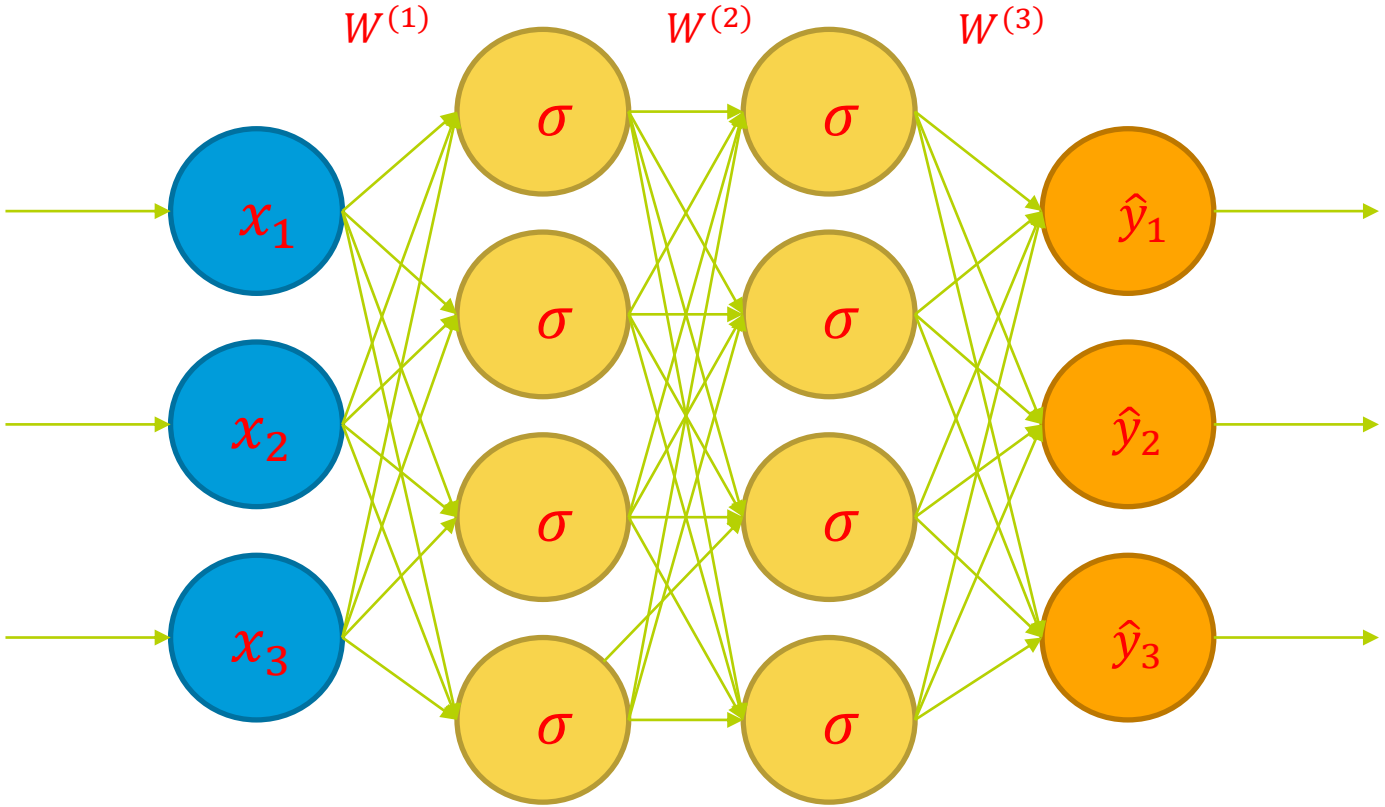
Hidden Layers



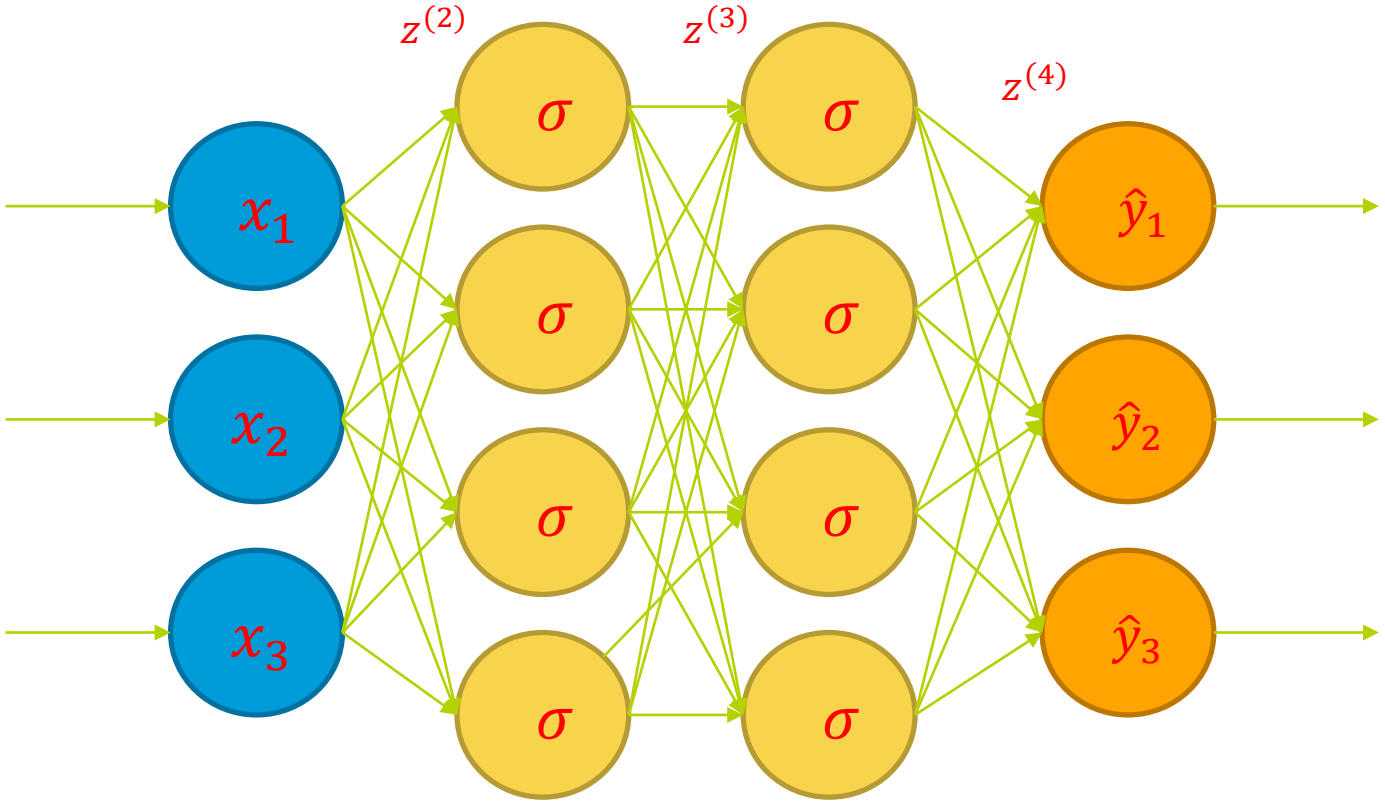
Output Layer



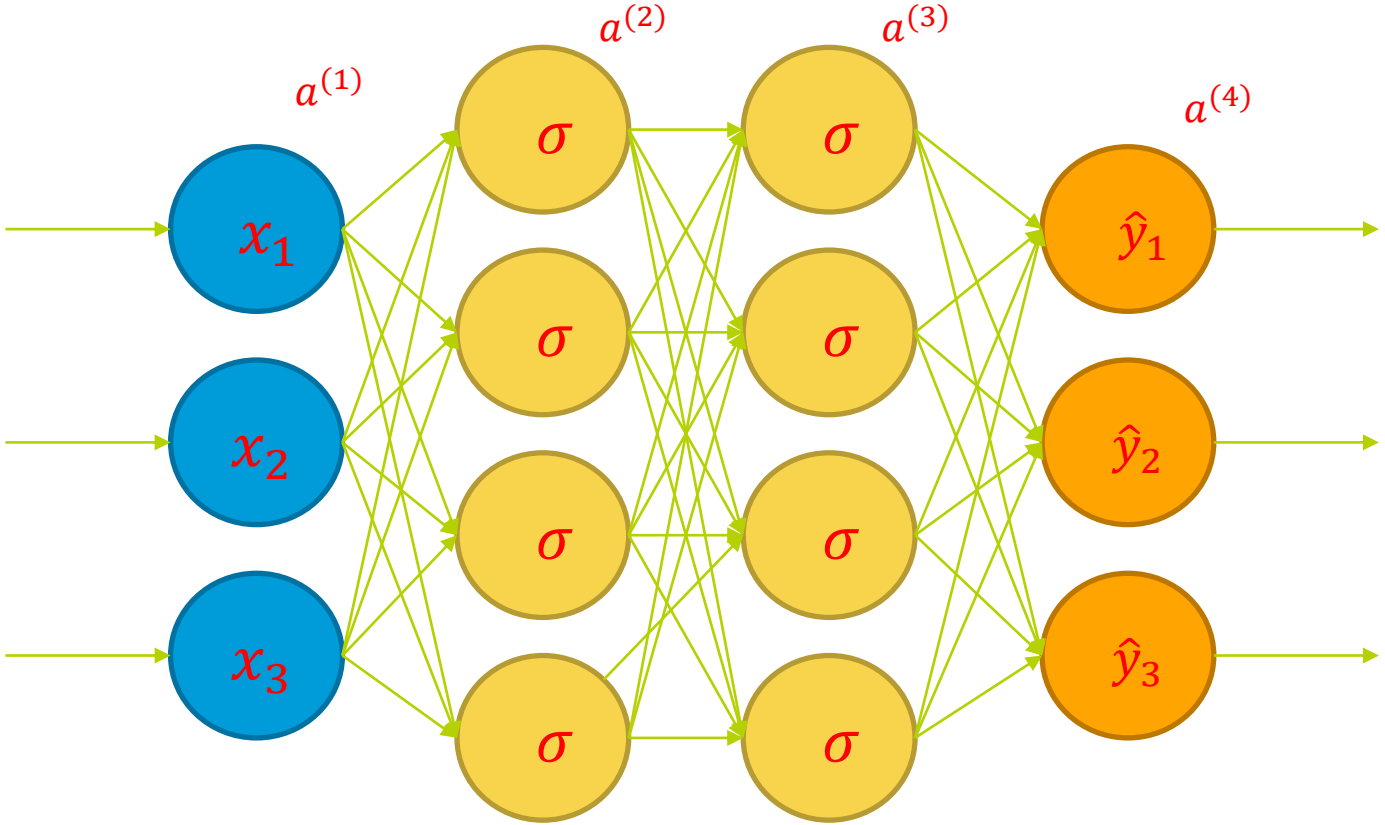
Weights (represented by matrices)



Net Input (sum of weighted inputs, before activation function)



Activations (output of neurons to next layer)



Matrix representation of computation

x

$(x = a^{(1)})$

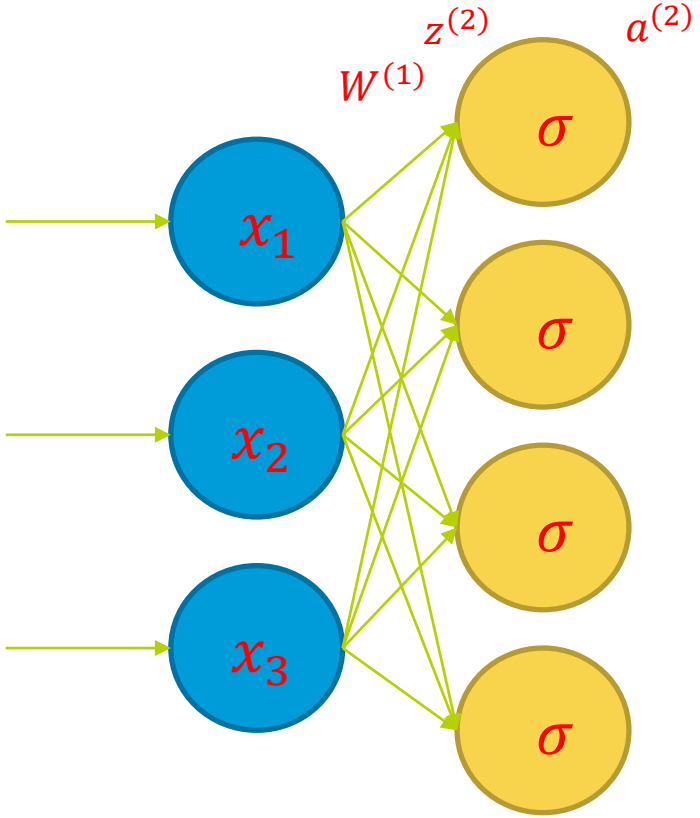
$z^{(2)} = xW^{(1)}$

$a^{(2)} = \sigma(z^{(2)})$

$W^{(1)}$ is a 3x4 matrix

$z^{(2)}$ is a 4-vector

$a^{(2)}$ is a 4-vector



Continuing the Computation

For a single training instance (data point)

Input: vector x (a row vector of length 3)
Output: vector $\hat{y}$ (a row vector of length 3)

$$z^{(2)} = xW^{(1)} \qquad a^{(2)} = \sigma(z^{(2)})$$

$$z^{(3)} = a^{(2)}W^{(2)} \qquad a^{(3)} = \sigma(z^{(3)})$$

$$z^{(4)} = a^{(3)}W^{(3)} \qquad \hat{y} = softmax(z^{(4)})$$

Multiple data points

In practice, we do these computation for many data points at the same time, by “stacking” the rows into a matrix.

But the equations look the same!

Input: matrix x (an nx3 matrix) (each row a single instance)
Output: vector $\hat{y}$ (an nx3 matrix) (each row a single prediction)

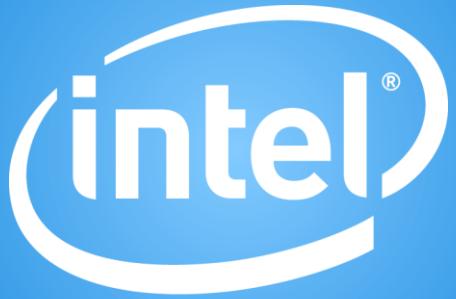
$$z^{(2)} = xW^{(1)} \qquad a^{(2)} = \sigma(z^{(2)})$$

$$z^{(3)} = a^{(2)}W^{(2)} \qquad a^{(3)} = \sigma(z^{(3)})$$

$$z^{(4)} = a^{(3)}W^{(3)} \qquad \hat{y} = softmax(z^{(4)})$$

Now we know how feedforward NNs do Computations.

Next, we will learn how to adjust the weights to learn from data.


intel
Software

Backpropagation in Neural Nets

Legal Notices and Disclaimers

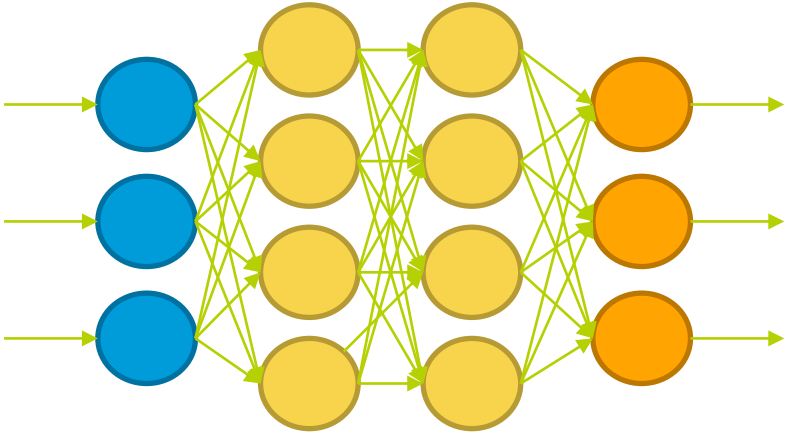
This presentation is for informational purposes only. INTEL MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at intel.com.

This sample source code is released under the Intel Sample Source Code License Agreement.

How to Train a Neural Net?

Input
(Feature Vector)



Output
(Label)

- Put in Training inputs, get the output
- Compare output to correct answers: Look at loss function J
- Adjust and repeat!
- Backpropagation tells us how to make a single adjustment using calculus.

How have we trained before?

- Gradient Descent!

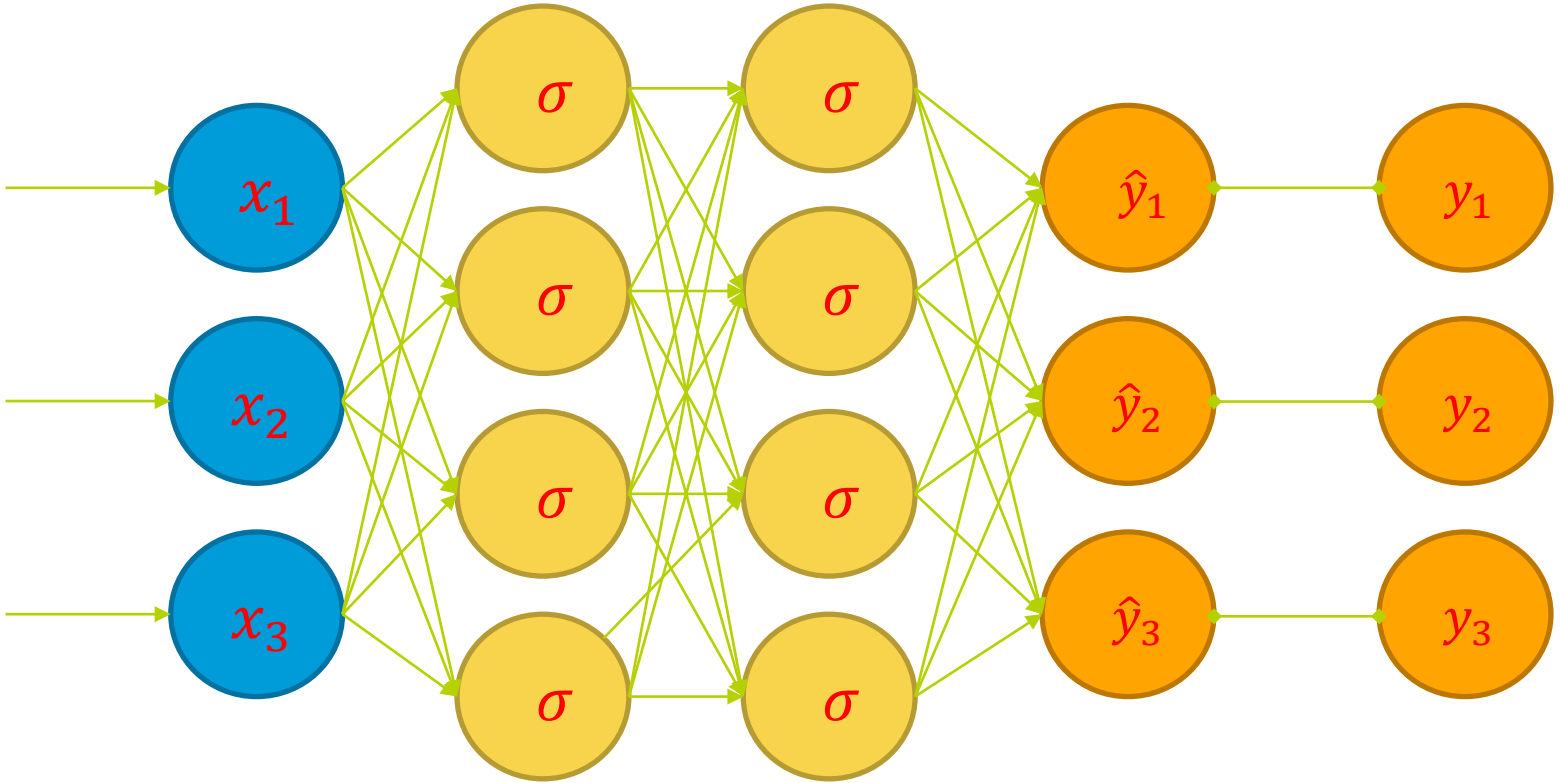
1. Make prediction
2. Calculate Loss
3. Calculate gradient of the loss function w.r.t. parameters
4. Update parameters by taking a step in the opposite direction
5. Iterate

How have we trained before?

- Gradient Descent!

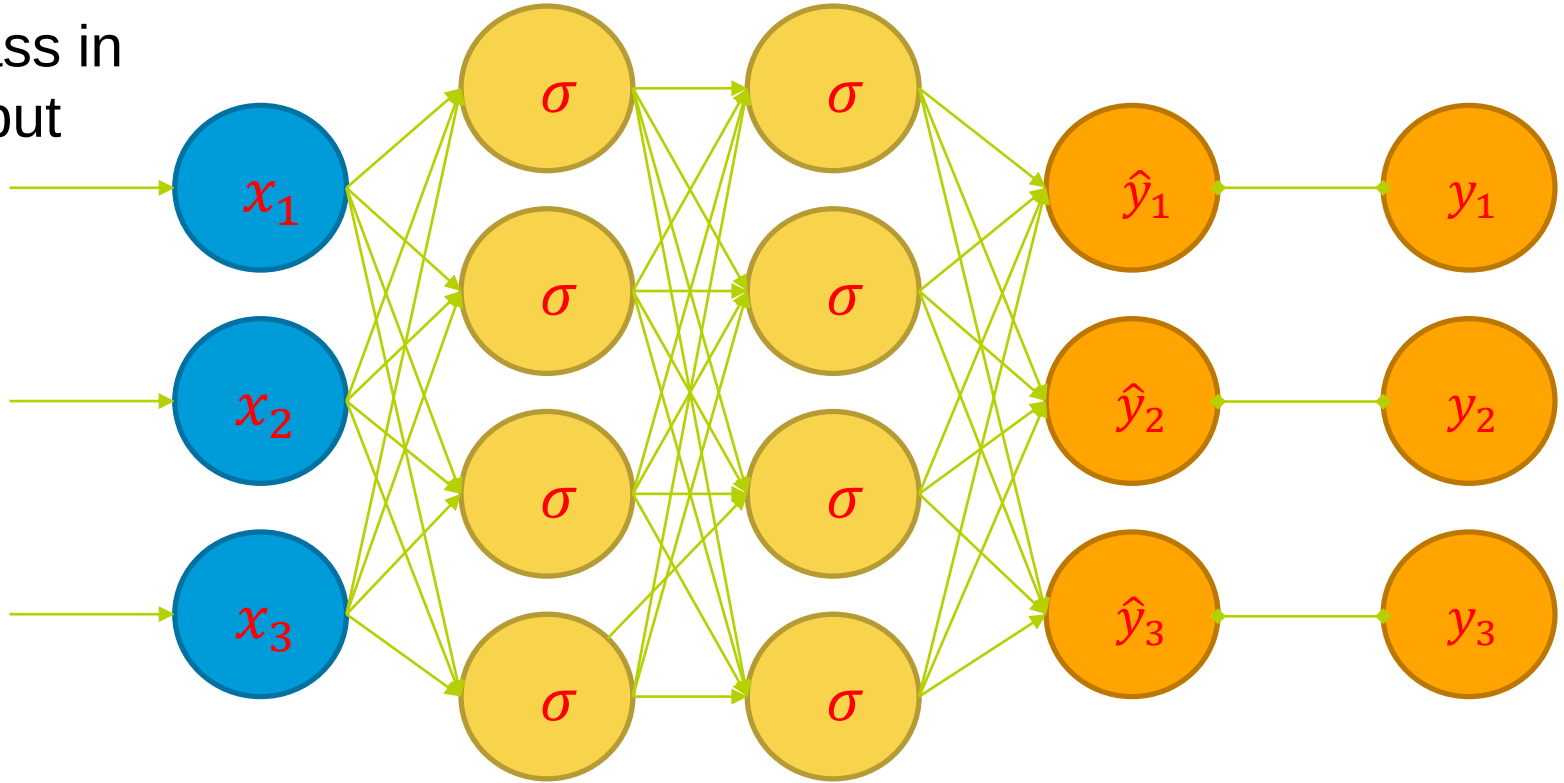
1. Make prediction
2. Calculate Loss
3. Calculate gradient of the loss function w.r.t. parameters
4. Update parameters by taking a step in the opposite direction
5. Iterate

Feedforward Neural Network



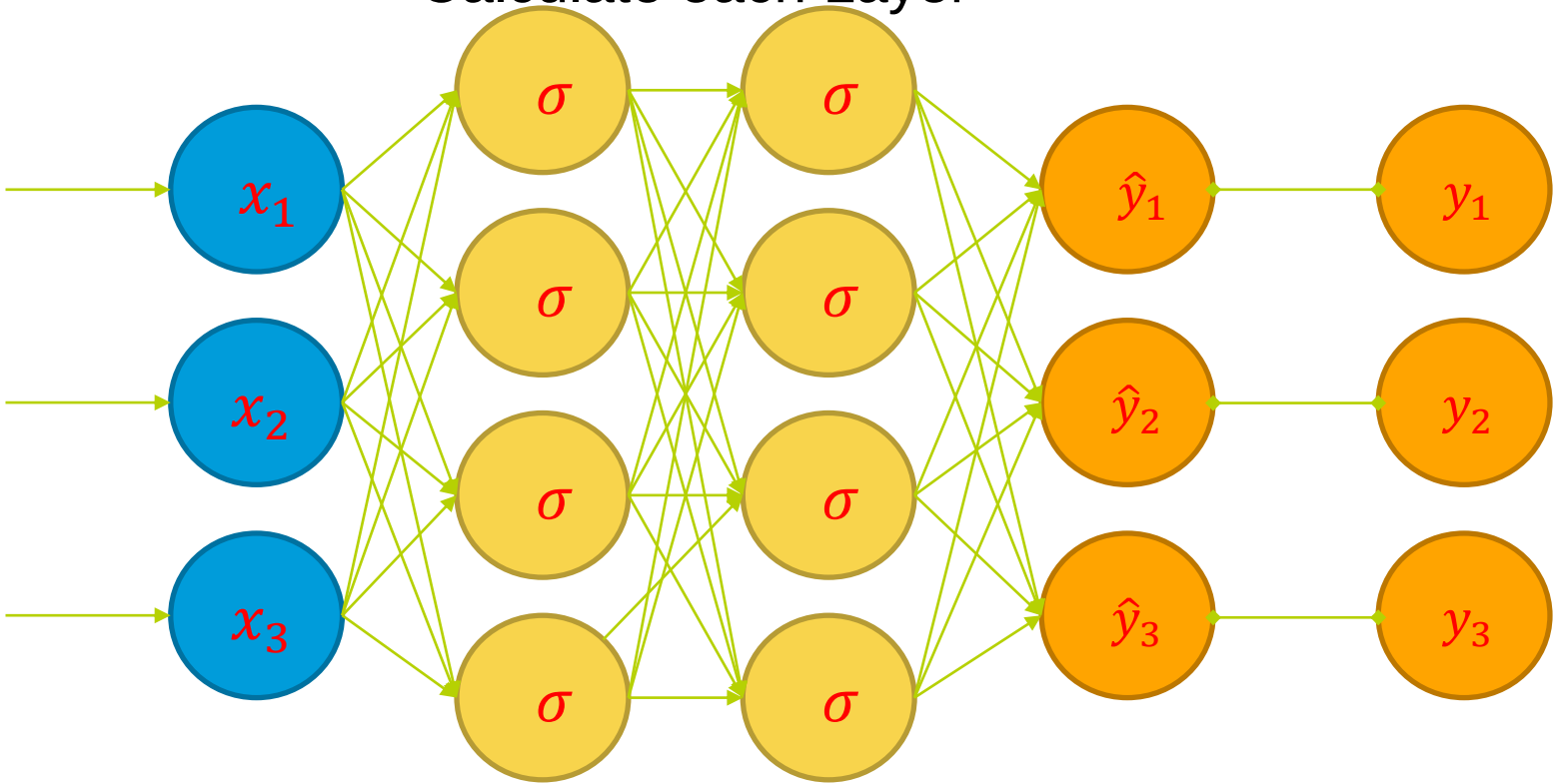
Forward Propagation

Pass in Input

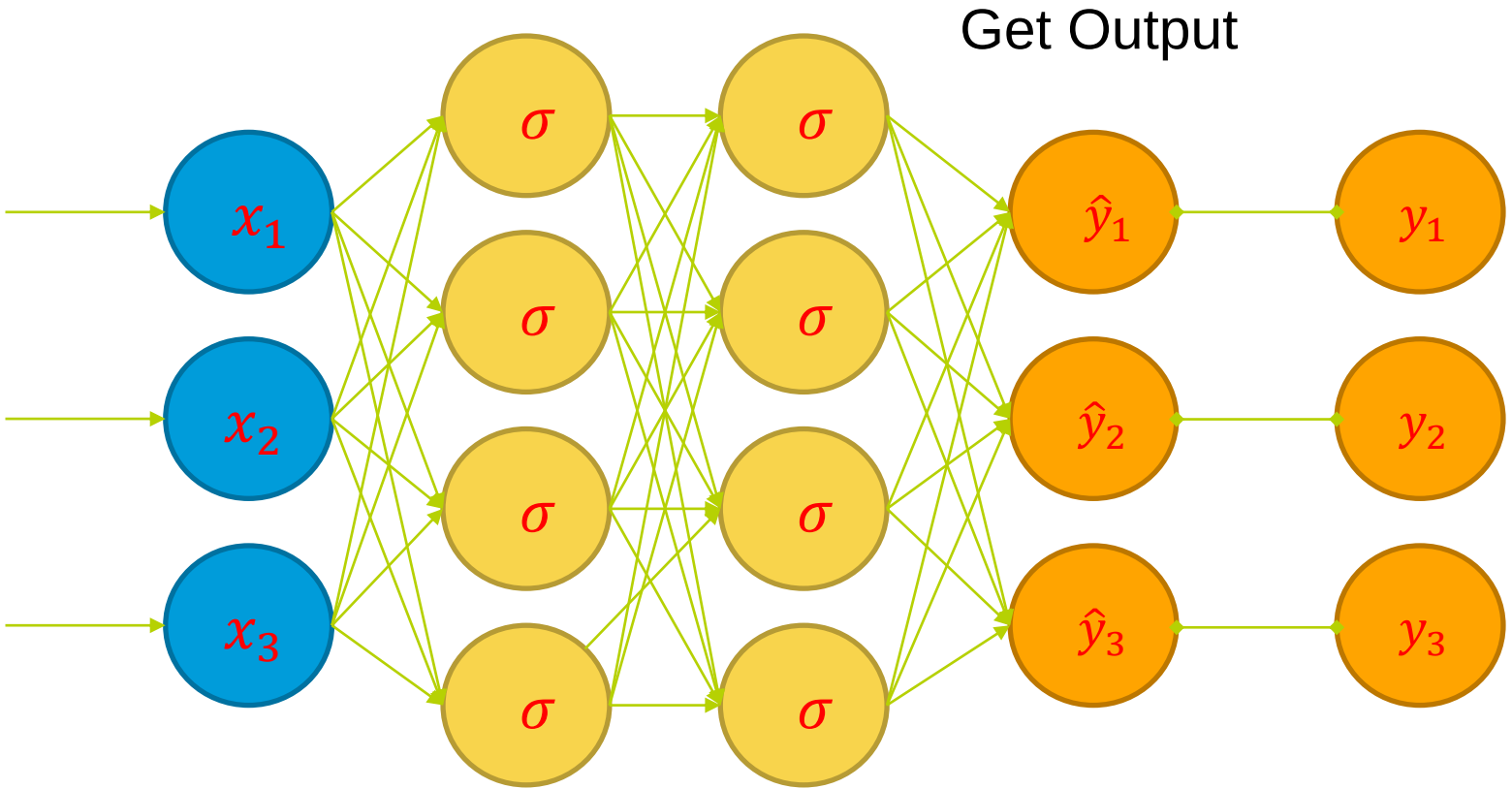


Forward Propagation

Calculate each Layer



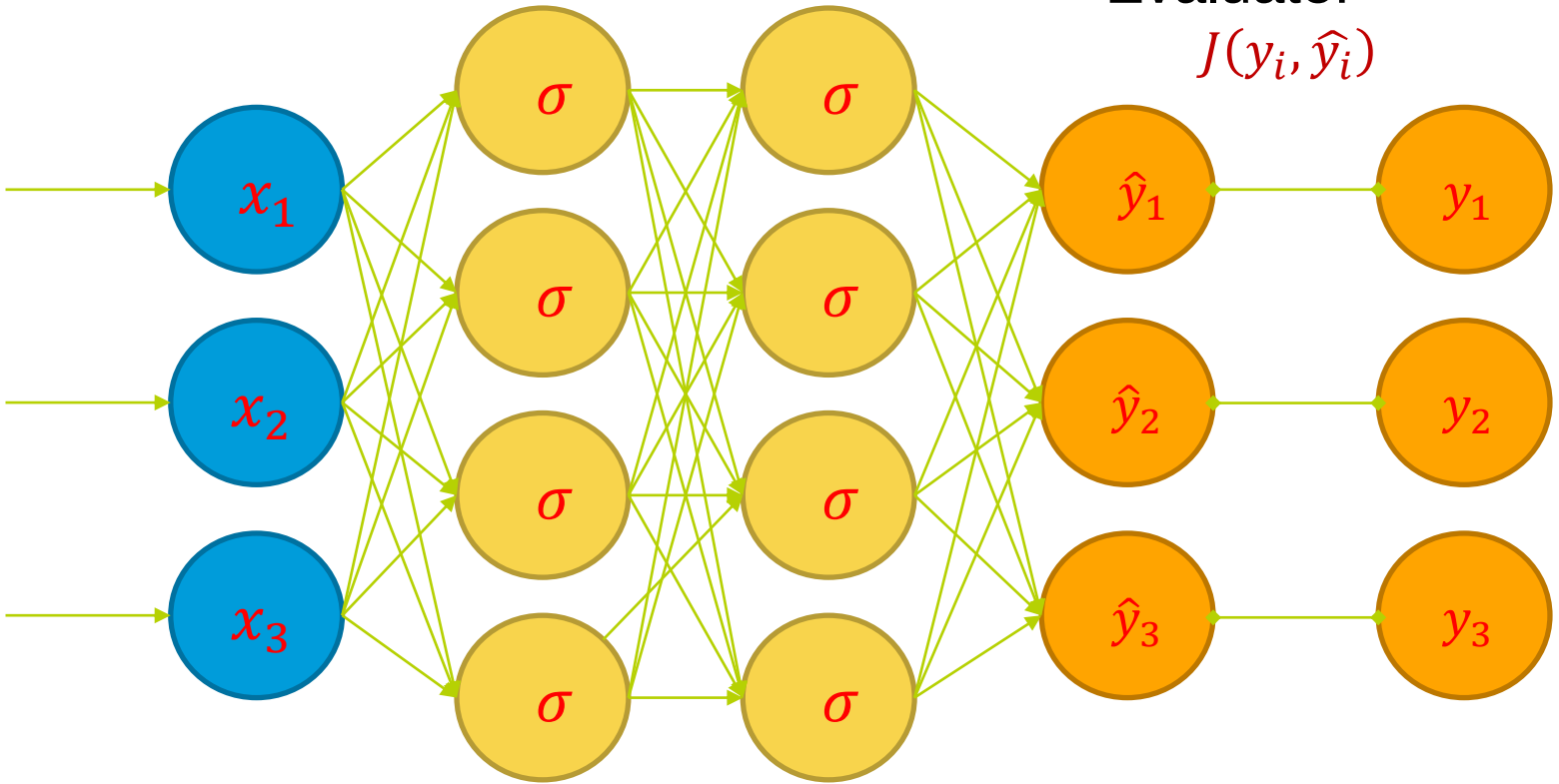
Forward Propagation



Forward Propagation

Evaluate:

$J(y_i, \hat{y}_i)$



How have we trained before?

- Gradient Descent!

1. Make prediction
2. Calculate Loss
3. Calculate gradient of the loss function w.r.t. parameters
4. Update parameters by taking a step in the opposite direction
5. Iterate

How to calculate gradient?

- Chain rule

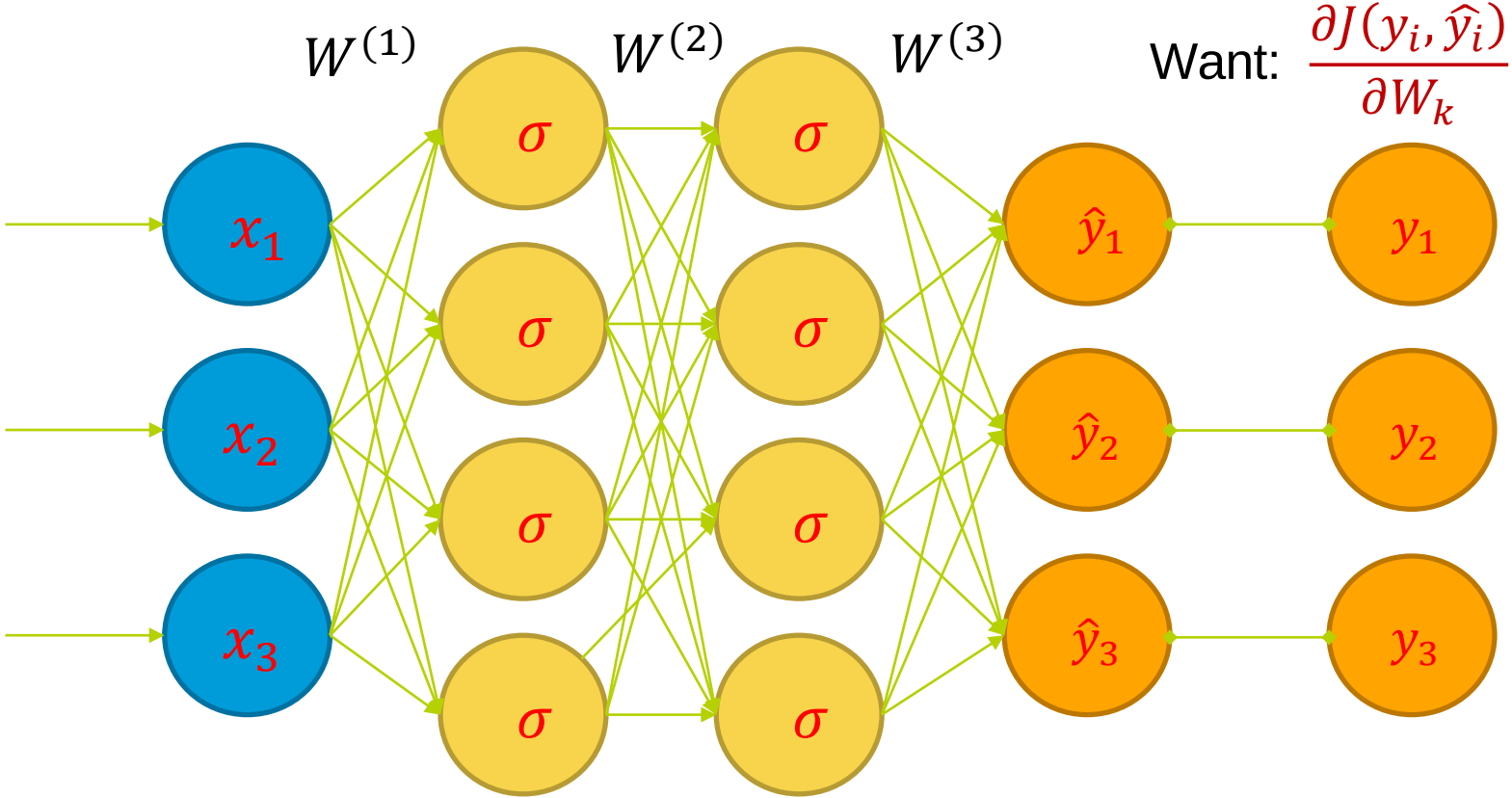
How to Train a Neural Net?

- How could we change the weights to make our Loss Function lower?
- Think of neural net as a function F: X -> Y
- F is a complex computation involving many weights W_k
- Given the structure, the weights “define” the function F (and therefore define our model)
- Loss Function is J(y,F(x))

How to Train a Neural Net?

- Get $\frac{\partial J}{\partial W_k}$ for every weight in the network.
- This tells us what direction to adjust each W_k if we want to lower our loss function.
- Make an adjustment and repeat!

Feedforward Neural Network



Calculus to the Rescue

- Use calculus, chain rule, etc. etc.
- Functions are chosen to have “nice” derivatives
- Numerical issues to be considered

Punchline

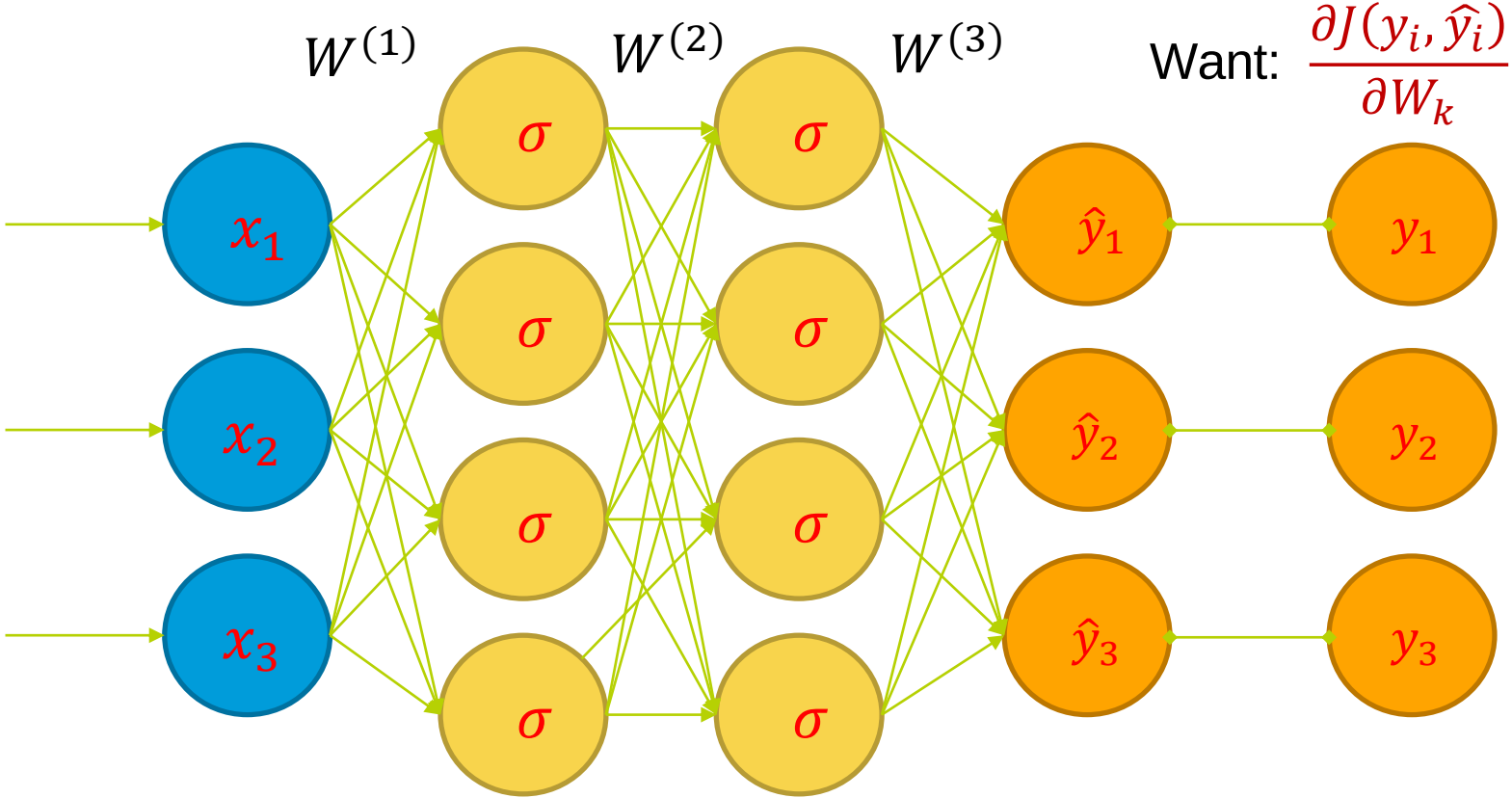
$$\frac{\partial J}{\partial W^{(3)}} = (\hat{y} - y) \cdot a^{(3)}$$

$$\frac{\partial J}{\partial W^{(2)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'\left(z^{(3)}\right) \cdot a^{(2)}$$

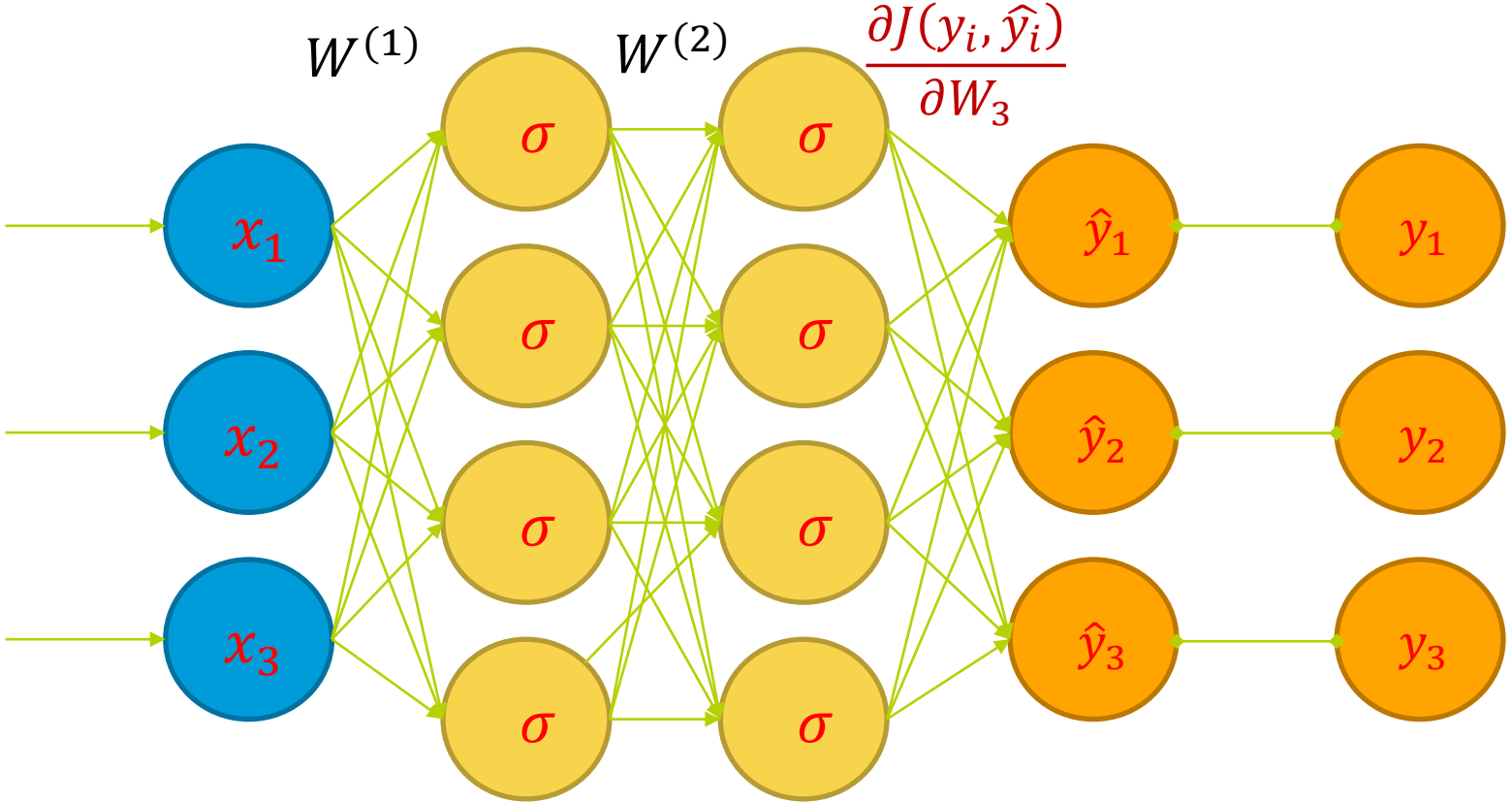
$$\frac{\partial J}{\partial W^{(1)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'\left(z^{(3)}\right) \cdot W^{(2)} \cdot \sigma'\left(z^{(2)}\right) \cdot X$$

- Recall that: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$
- Though they appear complex, above are easy to compute!

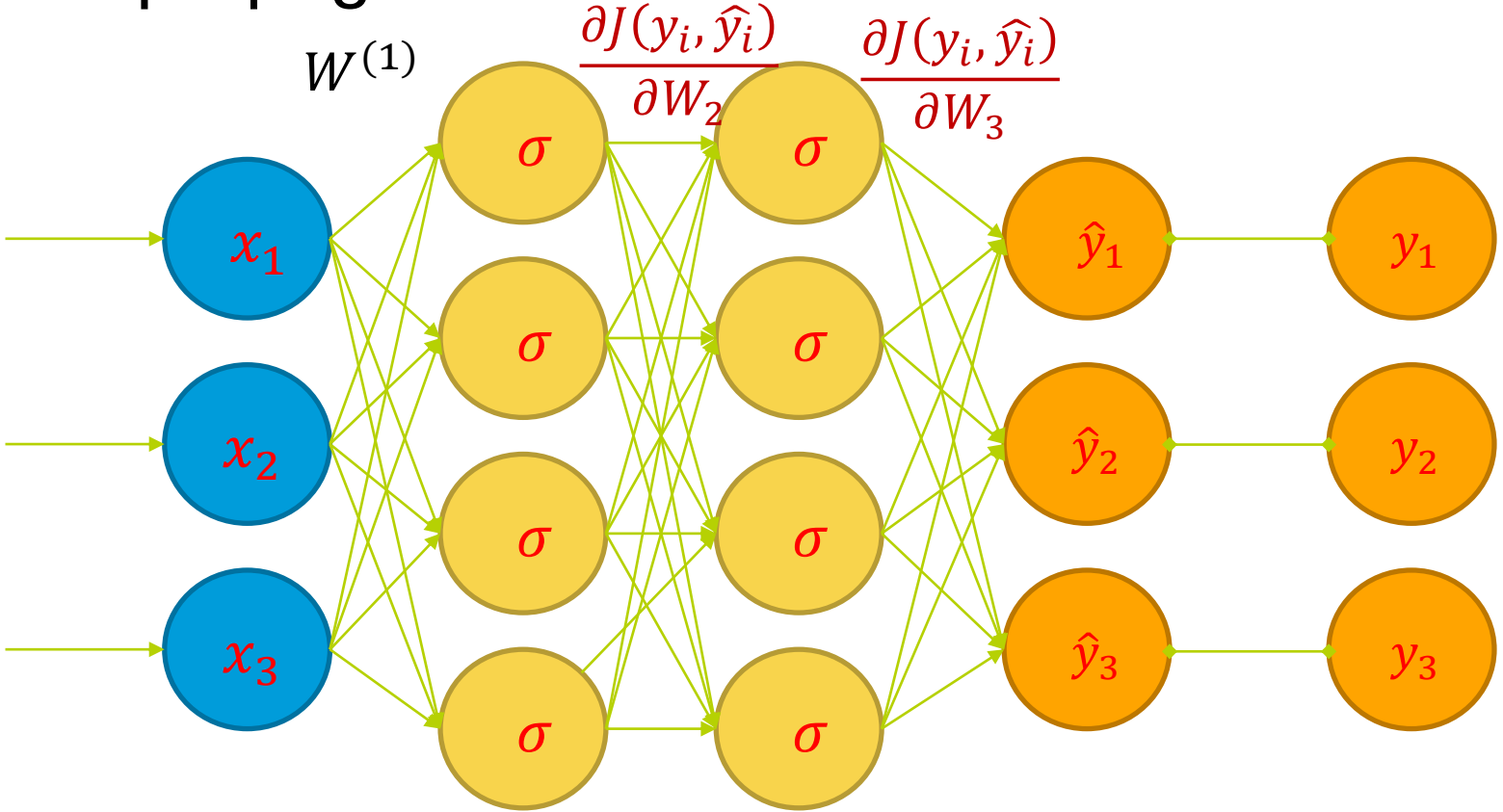
Backpropagation



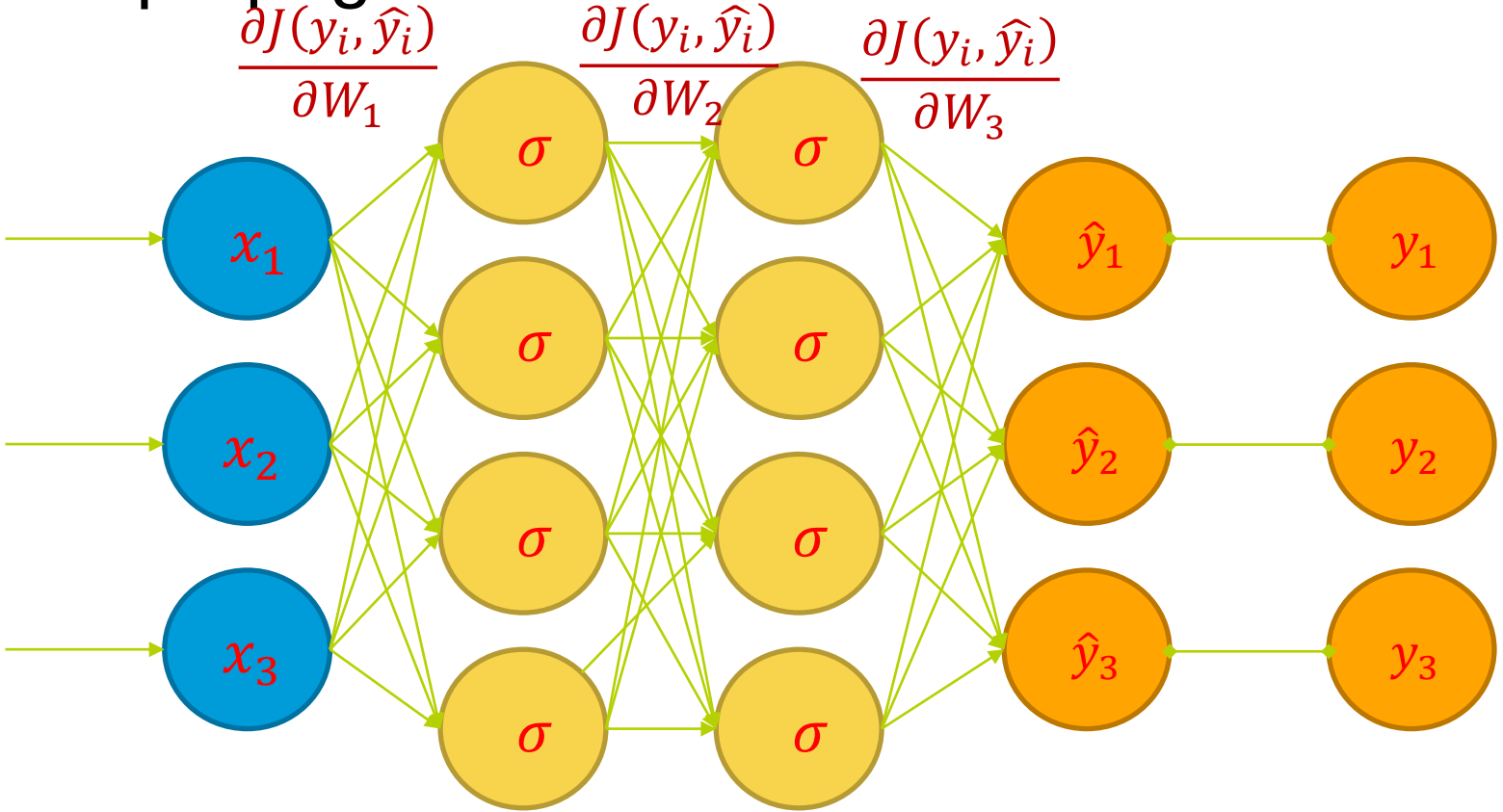
Backpropagation



Backpropagation



Backpropagation



How have we trained before?

- Gradient Descent!

1. Make prediction
2. Calculate Loss
3. Calculate gradient of the loss function w.r.t. parameters
4. Update parameters by taking a step in the opposite direction
5. Iterate

Vanishing Gradients

Recall that:

$$\frac{\partial J}{\partial W^{(1)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'\left(z^{(3)}\right) \cdot W^{(2)} \cdot \sigma'\left(z^{(2)}\right) \cdot X$$

- Remember: $\sigma'(z) = \sigma(z)\left(1 - \sigma(z)\right) \leq .25$
- As we have more layers, the gradient gets very small at the early layers.
- This is known as the “vanishing gradient” problem.
- For this reason, other activations (such as ReLU) have become more common.

Other Activation Functions

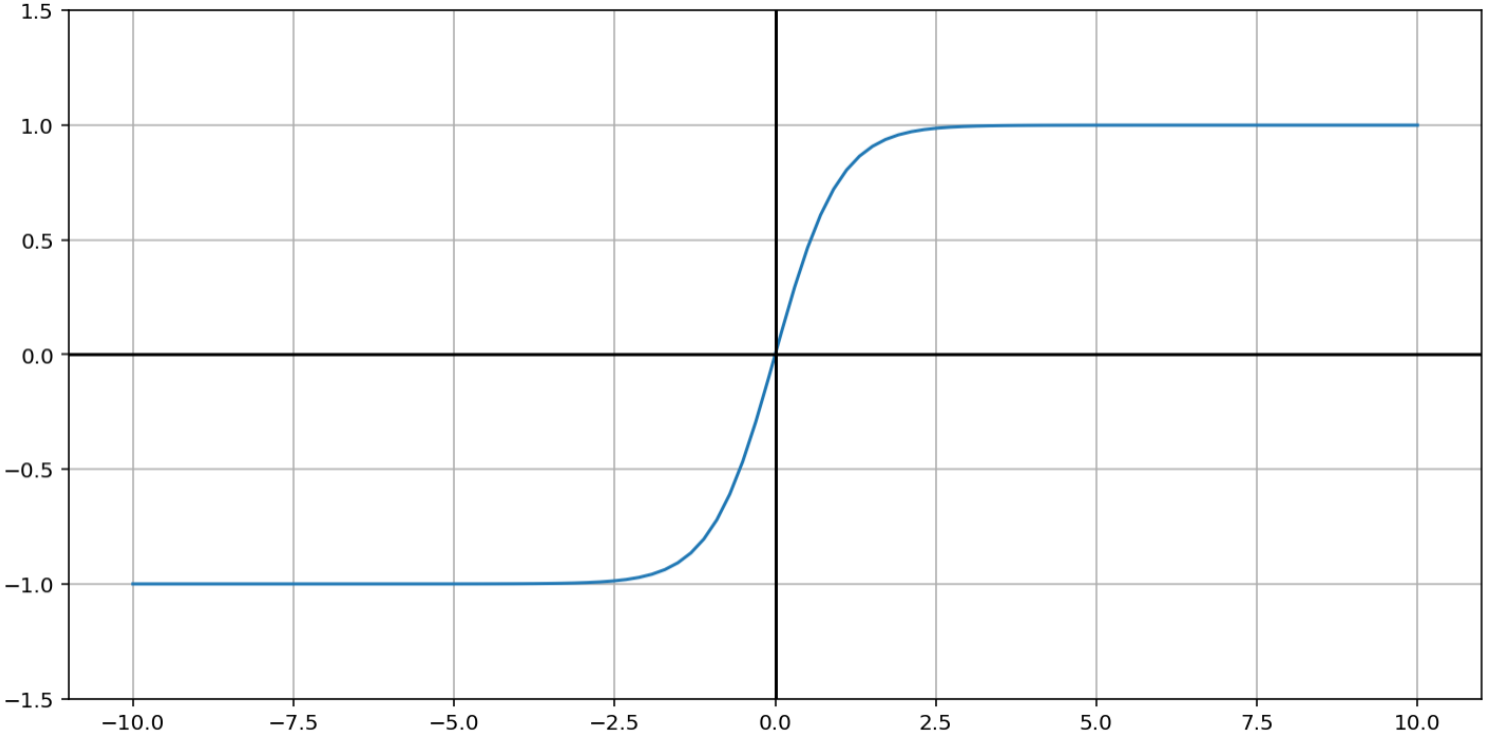
Hyperbolic Tangent Function

- Hyperbolic tangent function
- Pronounced “tanch”

$$tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

$$tanh(0) = 0$$
$$tanh(\infty) = 1$$
$$tanh(-\infty) = -1$$

Hyperbolic Tangent Function



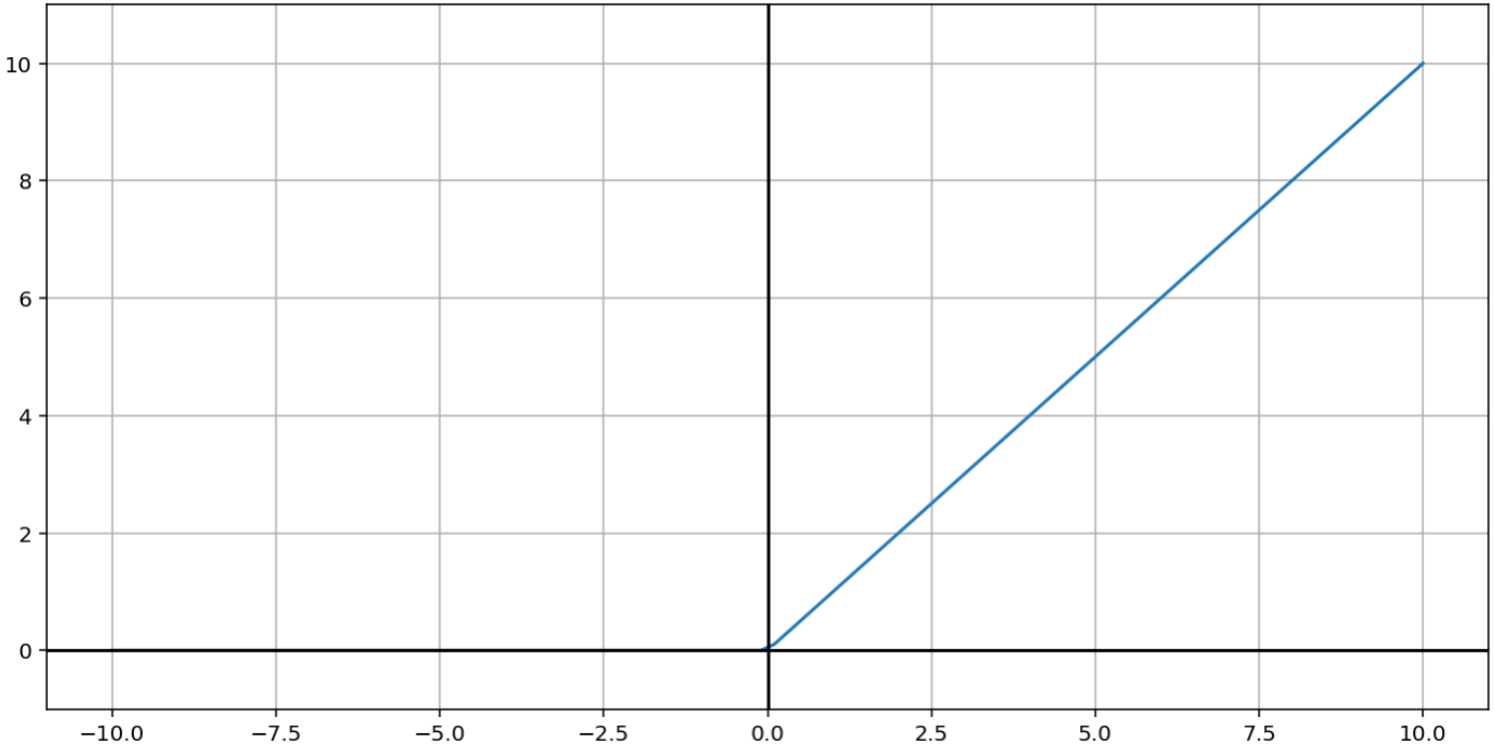
Rectified Linear Unit (ReLU)

$$ReLU(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$$

$$= \max(0, z)$$

$$ReLU(0) = 0$$
$$ReLU(z) = z \quad \text{for } (z \gg 0)$$
$$ReLU(-z) = 0$$

Rectified Linear Unit (ReLU)



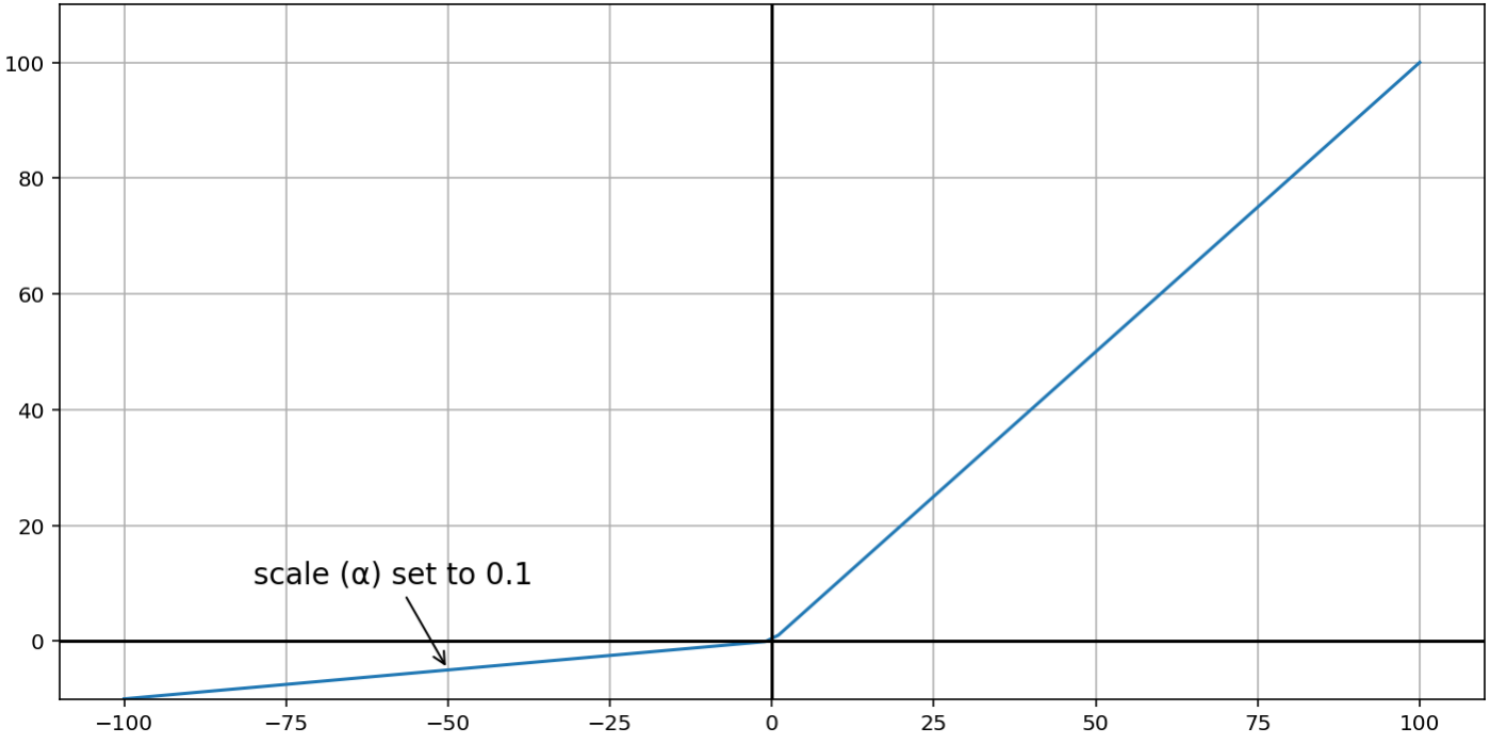
“Leaky” Rectified Linear Unit (ReLU)

$$LReLU(z) = \begin{cases} \alpha z, & z < 0 \\ z, & z \geq 0 \end{cases}$$

$$= \max(\alpha z, z) \qquad \text{for } (\alpha < 1)$$

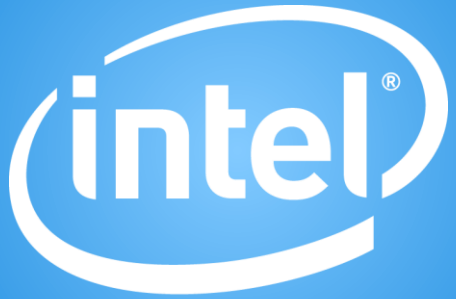
$$LReLU(0) = 0$$
$$LReLU(z) = z \qquad \text{for } (z \gg 0)$$
$$LReLU(-z) = -\alpha z$$

“Leaky” Rectified Linear Unit (ReLU)



What next?

- We now know how to make a single update to a model given some data.
- But how do we do the full training?
- We will dive into these details in the next lecture.


intel
Software

Details of Training Neural Nets

Legal Notices and Disclaimers

This presentation is for informational purposes only. INTEL MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at intel.com.

This sample source code is released under the Intel Sample Source Code License Agreement.

What next?

- Given an example (or group of examples), we know how to compute the derivative for each weight.
- How exactly do we update the weights?
- How often? (after each training data point? after all the training data points?)

What next? – Gradient Descent

- W_new = W_old - lr * derivative
- Classical approach – get derivative for entire data set, then take a step in that direction
- Pros: Each step is informed by all the data
- Cons: Very slow, especially as data gets big

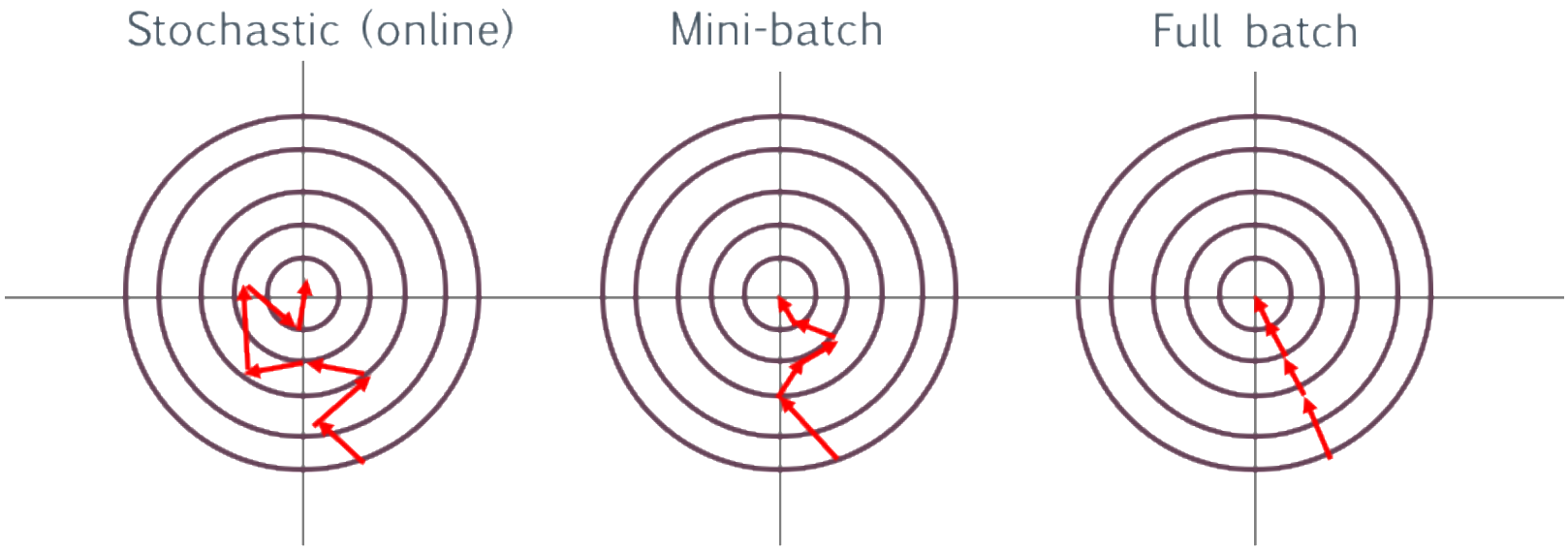
Another approach: Stochastic Gradient Descent

- Get derivative for just one point, and take a step in that direction
- Steps are “less informed” but you take more of them
- Should “balance out”
- Probably want a smaller step size
- Also helps “regularize”

Compromise approach: Mini-batch

- Get derivative for a ”small” set of points, then take a step in that direction
- Typical mini batch sizes are 16, 32
- Strikes a balance between two extremes

Comparison of Batching Approaches



Batching Terminology

- Full-batch: Use entire data set to compute gradient before updating
- Mini-batch: Use a smaller portion of data (but more than single example) to compute gradient before updating
- Stochastic Gradient Descent (SGD): Use a single example to compute gradient before updating (though sometimes people use SGD to refer to minibatch, also)

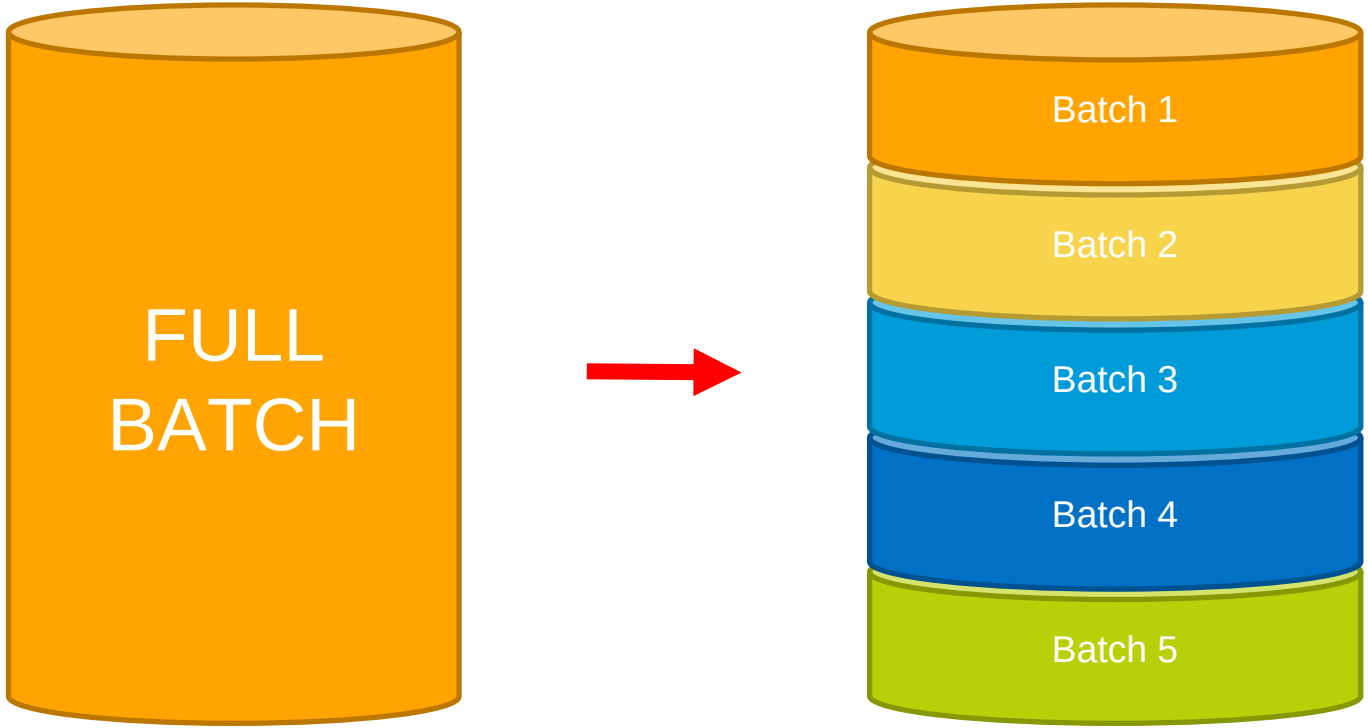
Batching Terminology

- An Epoch refers to a single pass through all of the training data.
- In full batch gradient descent, there would be one step taken per epoch.
- In SGD / Online learning, there would be n steps taken per epoch (n = training set size)
- In Minibatch there would be (n/batch size) steps taken per epoch
- When training, it is common to refer to the number of epochs needed for the model to be “trained”.

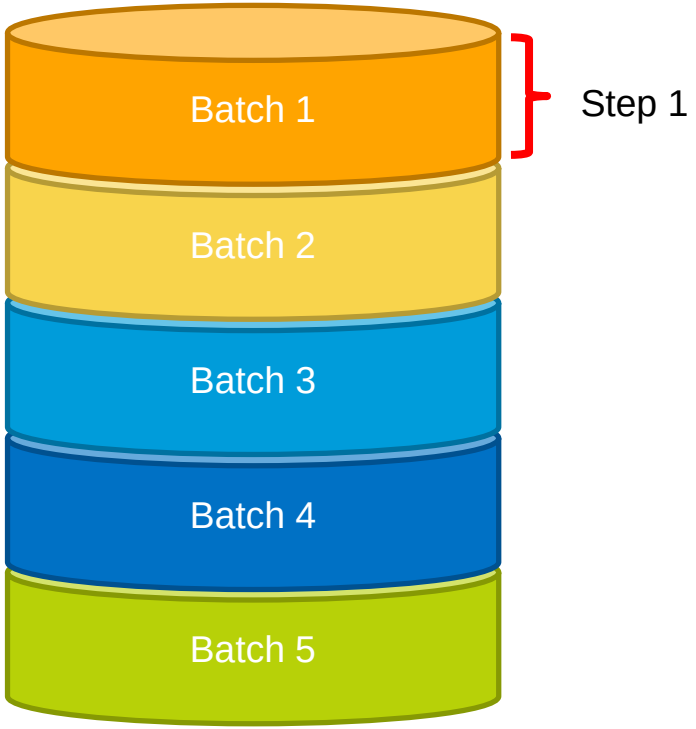
Note on Data Shuffling

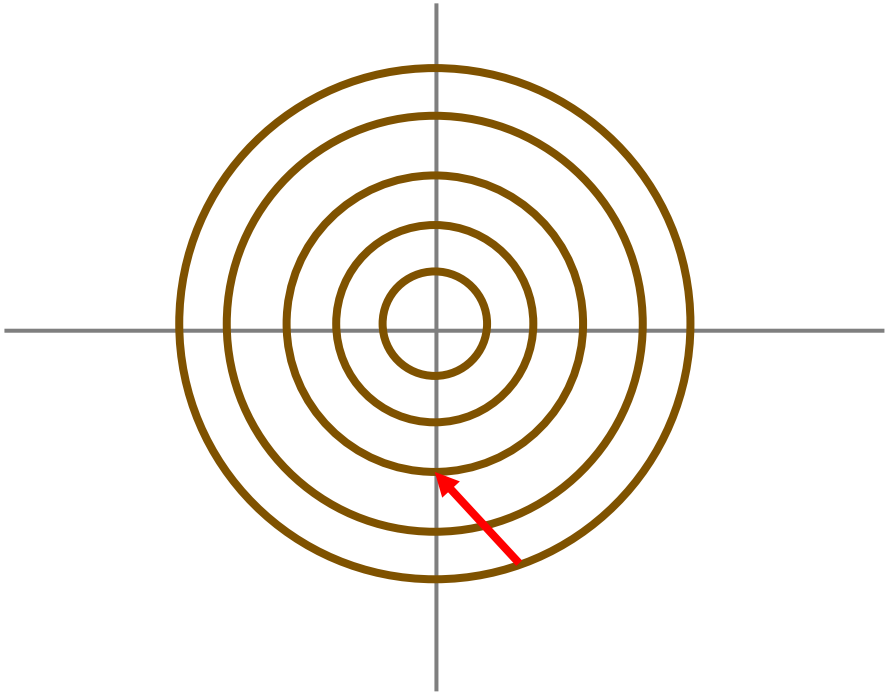
- To avoid any cyclical movement and aid convergence, it is recommended to shuffle the data after each epoch.
- This way, the data is not seen in the same order every time, and the batches are not the exact same ones.

Feedforward Neural Network

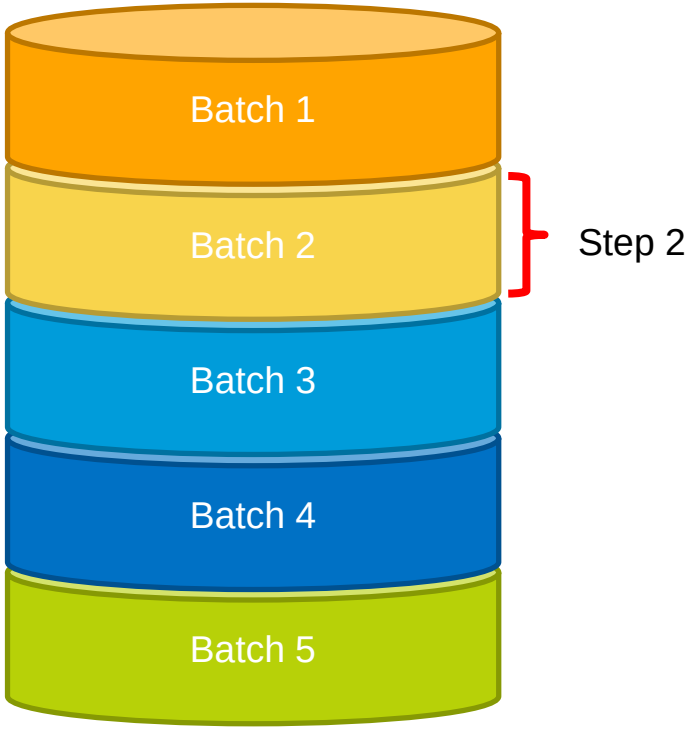


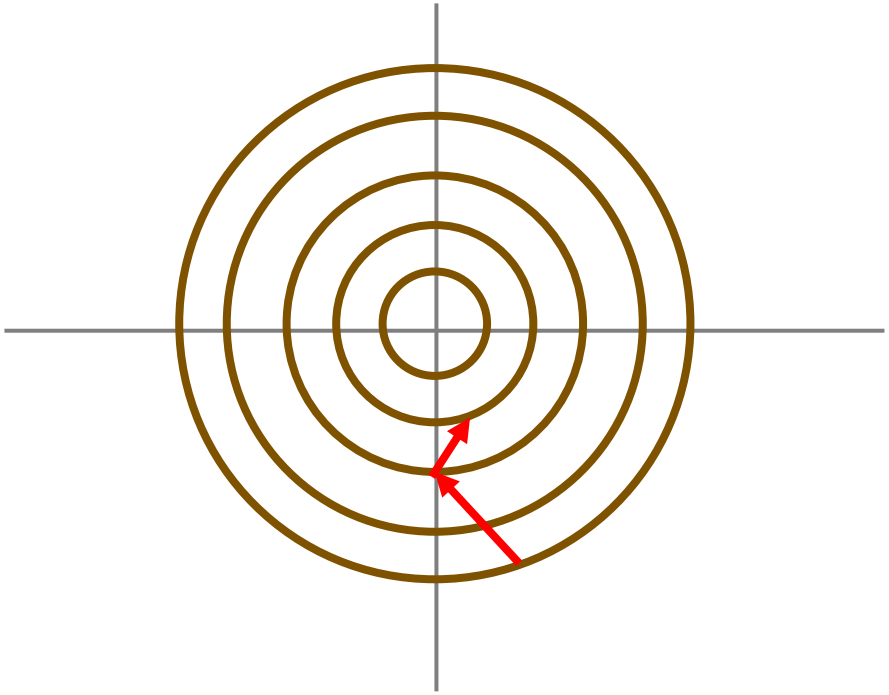
Training in Action



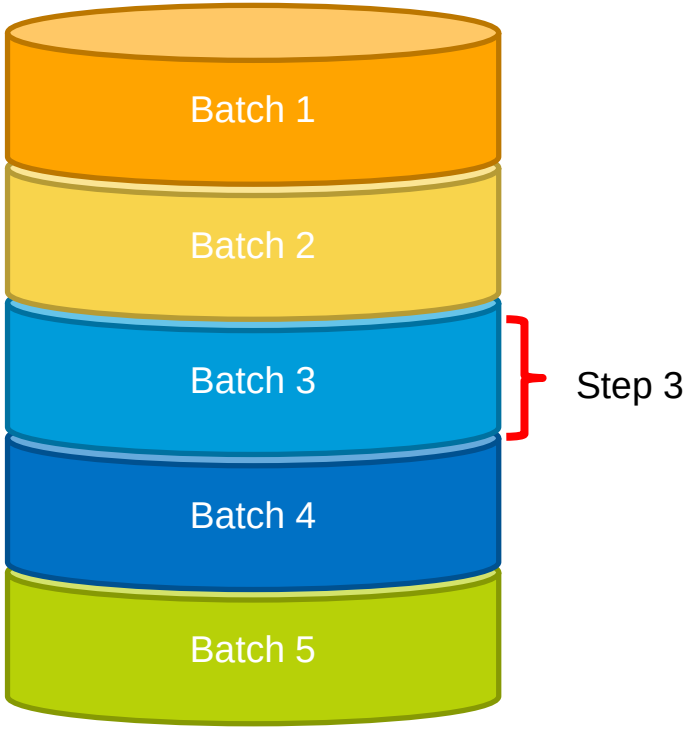


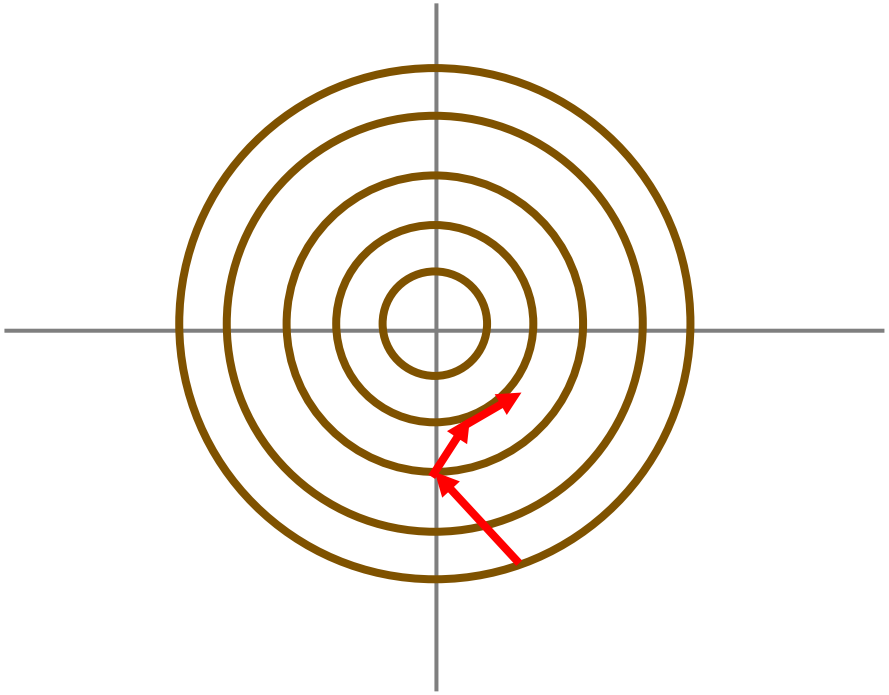
Training in Action



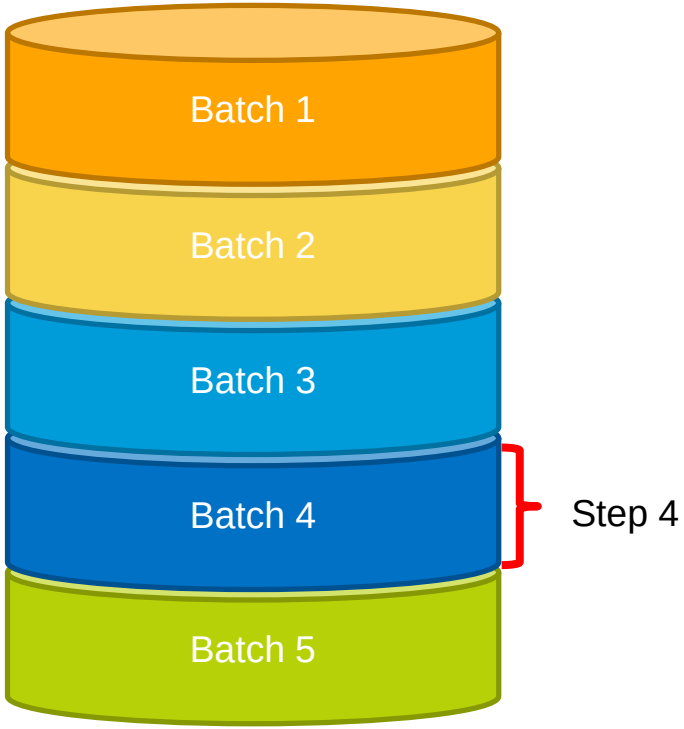


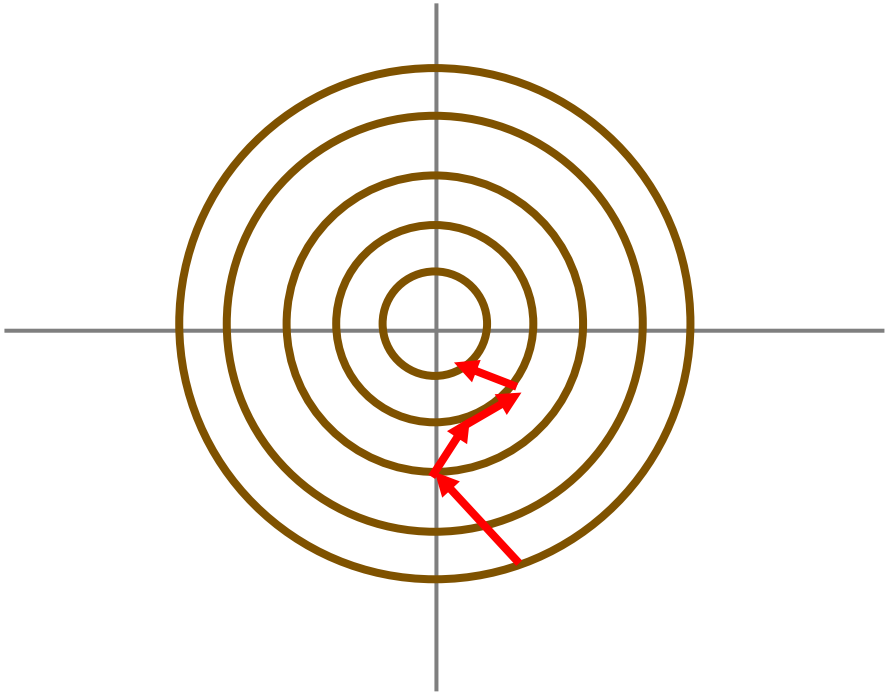
Training in Action



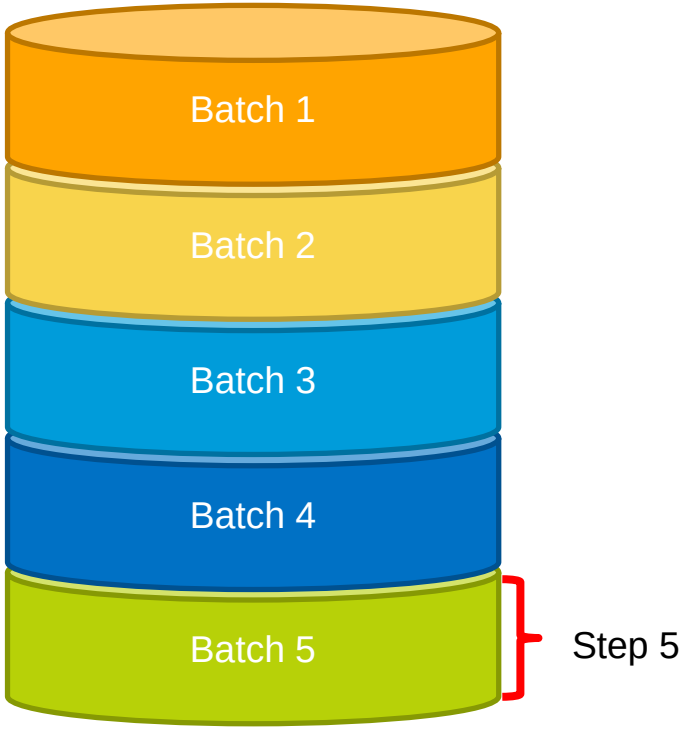


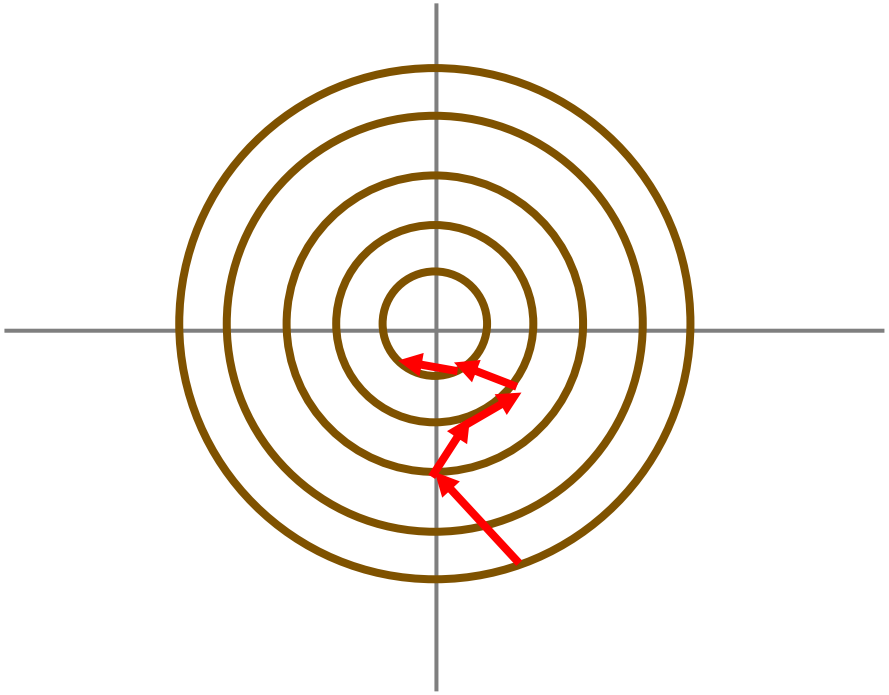
Training in Action



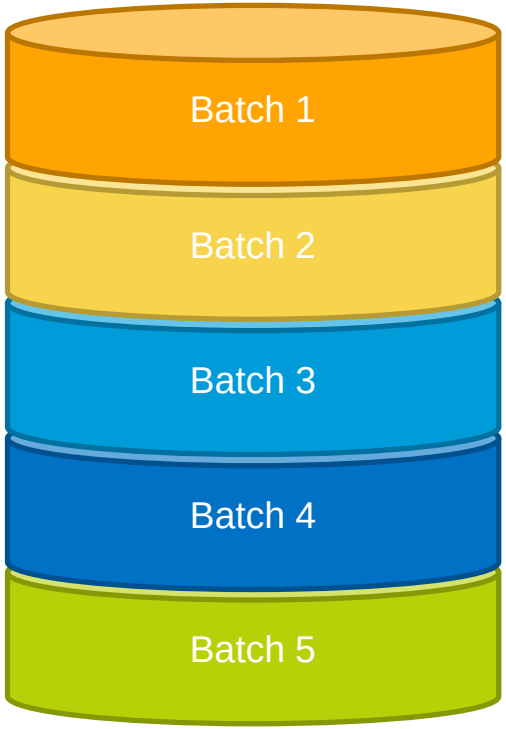


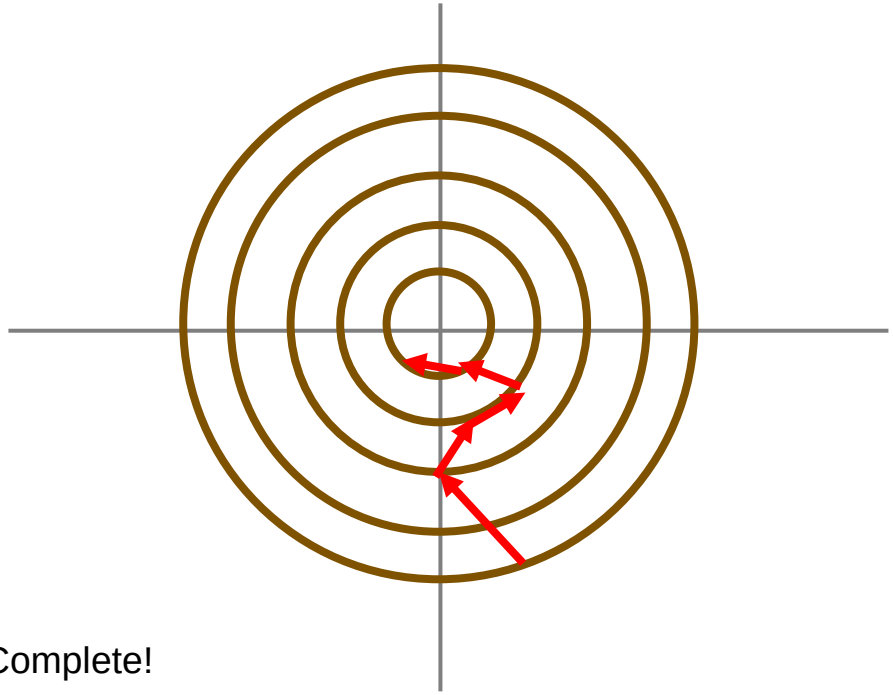
Training in Action





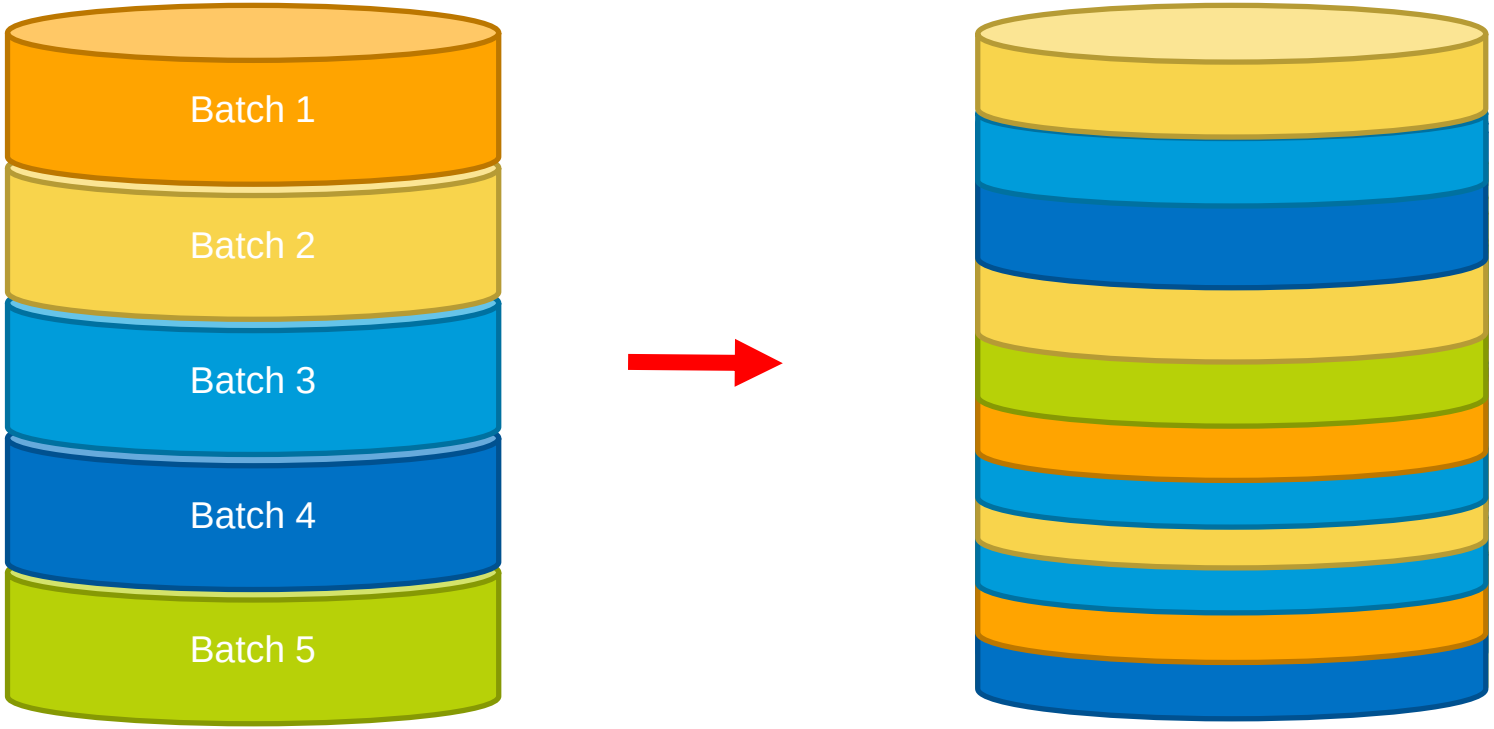
Training in Action



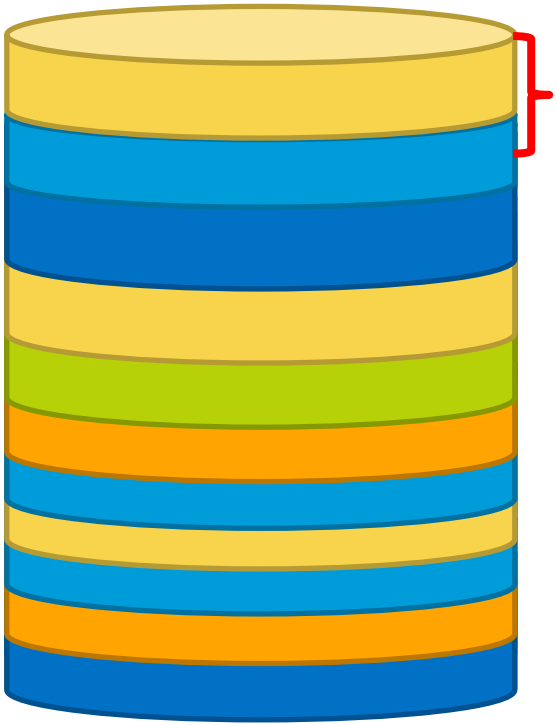


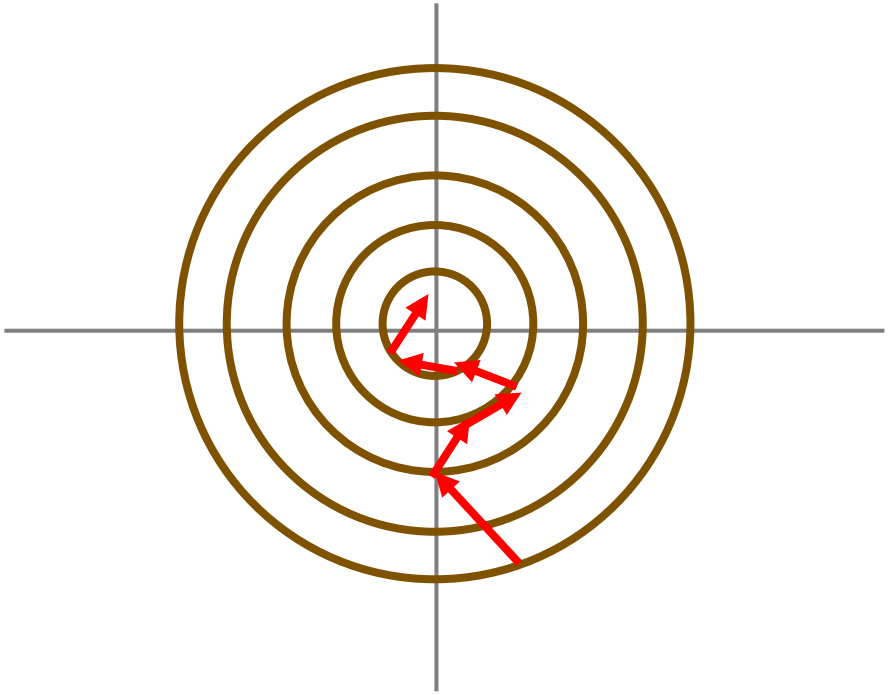
First Epoch Complete!

Shuffle the Data!



Shuffle the Data!





The Keras Package

- Keras allows easy construction, training, and execution of Deep Neural Networks
- Written in Python, and allows users to configure complicated models directly in Python
- Uses either Tensorflow or Theano “under the hood”
- Uses either CPU or GPU for computation
- Uses numpy data structures, and a similar command structure to scikit-learn (model.fit , model.predict, etc.)

Typical Command Structure in Keras

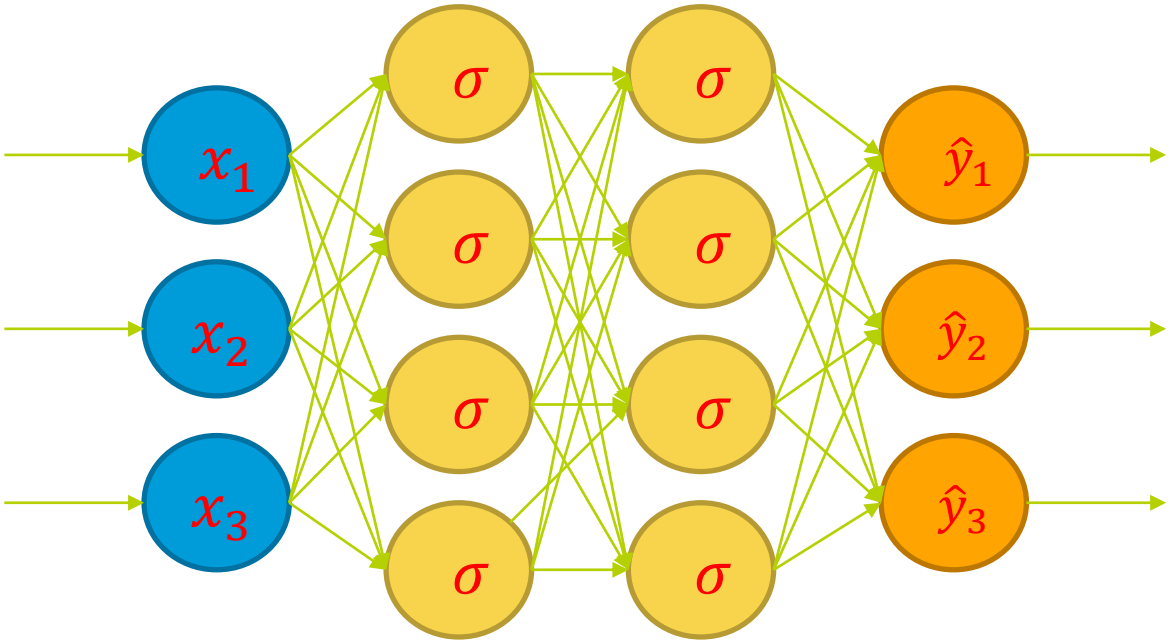
- Build the structure of your network.
- Compile the model, specifying your loss function, metrics, and optimizer (which includes the learning rate).
- Fit the model on your training data (specifying batch size, number of epochs)
- Predict on new data
- Evaluate your results

Building the model

- Keras provides two approaches to building the structure of your model:
- Sequential Model: allows a linear stack of layers – simpler and more convenient if model has this form
- Functional API: more detailed and complex, but allows more complicated architectures
- We will focus on the Sequential Model.

Running Example, this time in Keras

Let's build this Neural Network structure shown below in Keras:



Keras - Sequential Model

First, import the Sequential function and initialize your model object:

```
from keras.models import Sequential
model = Sequential()
```

Keras - Sequential Model

Then we add layers to the model one by one.

```
from keras.layers import Dense, Activation

# For the first layer, specify the input dimension
model.add(Dense(units=4, input_dim=3))

# Specify an activation function
model.add(Activation(sigmoid'))

# For subsequent layers, the input dimension is presumed from
# the previous layer
model.add(Dense(units=4))
model.add(Activation(sigmoid'))
model.add(Dense(units=3))
model.add(Activation('softmax'))
```

Multiclass Classification with Neural Networks

- For binary classification problems, we have a final layer with a single node and a sigmoid activation.
- This has many desirable properties
  - Gives an output strictly between 0 and 1
  - Can be interpreted as a probability
  - Derivative is “nice”
  - Analogous to logistic regression
- Is there a natural extension of this to a multiclass setting?

Multiclass Classification with Neural Networks

- Reminder: one hot encoding for categories
- Take a vector with length equal to the number of categories
- Represent each category with one at a particular position (and zero everywhere else)

$$\begin{bmatrix}1\\0\\0\end{bmatrix} \quad \begin{bmatrix}0\\1\\0\end{bmatrix} \quad \begin{bmatrix}0\\0\\1\end{bmatrix}$$

Cat Dog Toaster

Multiclass Classification with Neural Networks

- For multiclass classification problems, let the final layer be a vector with length equal to the number of possible classes.
- Extension of sigmoid to multiclass is the softmax function.
- $softmax(z_i) = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}}$
- Yields a vector with entries that are between 0 and 1, and sum to 1

Multiclass Classification with Neural Networks

- For loss function use “categorical cross entropy”
- This is just the log-loss function in disguise

$$C.E. = -\sum_{i=1}^{n} y_i \log(\hat{y}_i)$$

- Derivative has a nice property when used with softmax

$$\frac{\partial C.E.}{\partial softmax} \cdot \frac{\partial softmax}{\partial z_i} = \hat{y}_i - y_i$$

Ways to scale inputs

- Linear scaling to the interval [0,1]

$$x_i = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

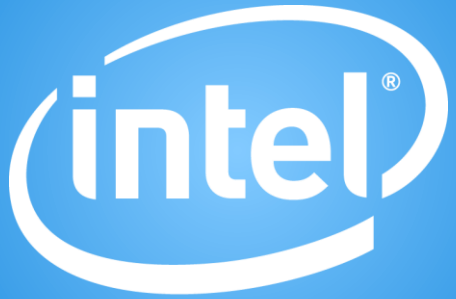
- Linear scaling to the interval [-1,1]

$$x_i = 2\left(\frac{x_i - \bar{x}}{x_{max} - x_{min}}\right) - 1$$

Ways to scale inputs

- Standardization (making variable approx. std. normal)

$$x_i = \frac{x_i - \bar{x}}{\sigma}; \qquad \sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$


intel
Software

Regularization Techniques for Deep Learning

Legal Notices and Disclaimers

This presentation is for informational purposes only. INTEL MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at intel.com.

This sample source code is released under the Intel Sample Source Code License Agreement.

Regularizing Neural Networks

We have several means by which to help “regularize” neural networks – that is, to prevent overfitting

- Regularization penalty in cost function
- Dropout
- Early stopping
- Stochastic / Mini-batch Gradient descent (to some degree)

Penalized Cost function

- One option is to explicitly add a penalty to the loss function for having high weights.
- This is a similar approach to Ridge Regression

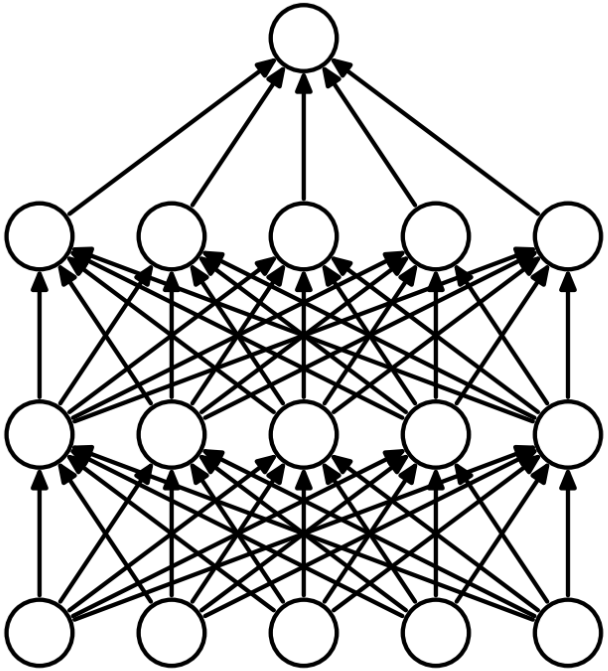
$$J = \frac{1}{2n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 + \lambda \sum_{j=1}^{m} W_i^2$$

- Can have an analogous expression for Categorical Cross Entropy

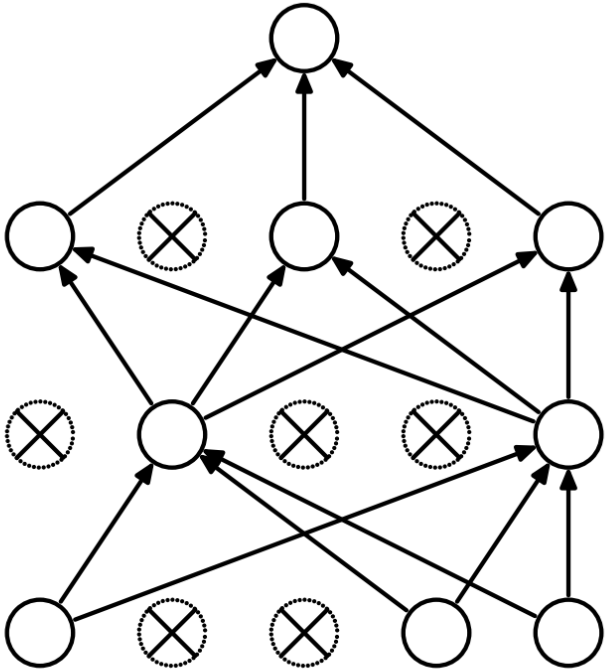
Dropout

- Dropout is a mechanism where at each training iteration (batch) we randomly remove a subset of neurons
- This prevents the neural network from relying too much on individual pathways, making it more “robust”
- At test time we “rescale” the weight of the neuron to reflect the percentage of the time it was active

Dropout - Visualization



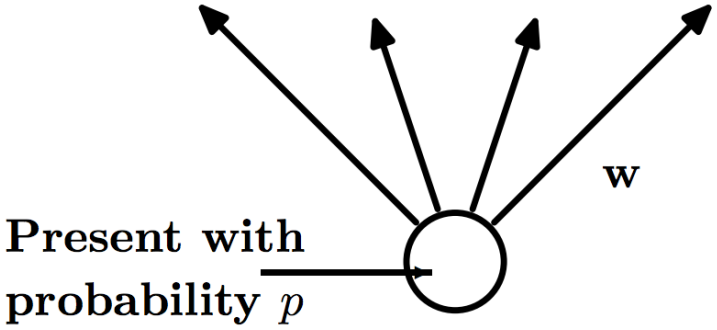
(a) Standard Neural Net



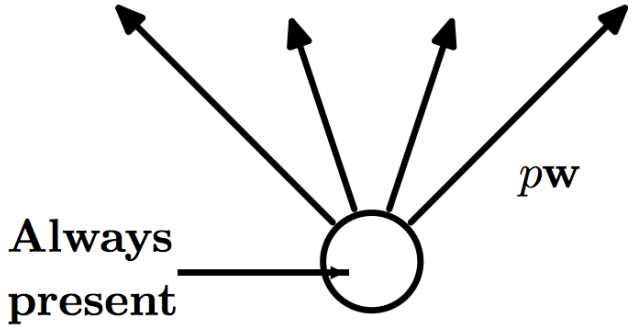
(b) After applying dropout.

Dropout - Visualization

- If the neuron was present with probability *p*, at test time we scale the outbound weights by a factor of *p*.



(a) At training time



(b) At test time

Early Stopping

- Another, more heuristical approach to regularization is early stopping.
- This refers to choosing some rules after which to stop training.
- Example:
  - Check the validation log-loss every 10 epochs.
  - If it is higher than it was last time, stop and use the previous model (i.e. from 10 epochs previous)

Optimizers

- We have considered approaches to gradient descent which vary the number of data points involved in a step.
- However, they have all used the standard update formula:

$$W := W - \alpha \cdot \nabla J$$

- There are several variants to updating the weights which give better performance in practice.
- These successive “tweaks” each attempt to improve on the previous idea.
- The resulting (often complicated) methods are referred to as “optimizers”.

Momentum

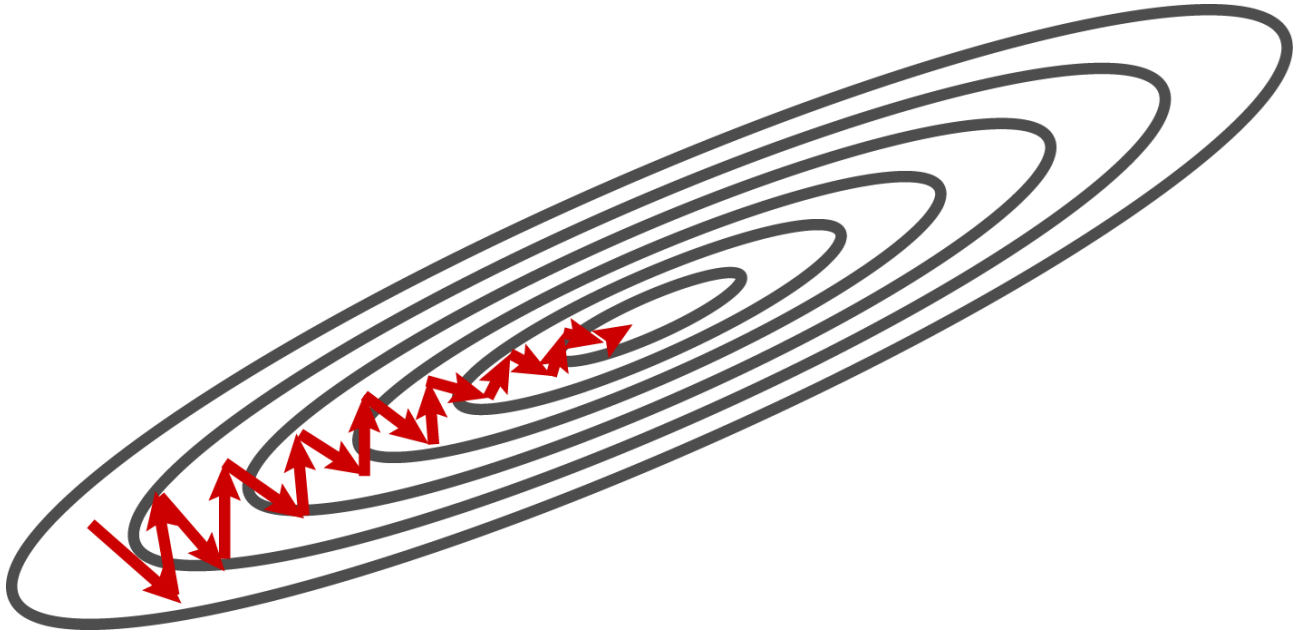
- Idea, only change direction by a little bit each time.
- Keeps a “running average” of the step directions, smoothing out the variation of the individual points.

$$v_t := \eta \cdot v_{t-1} - \alpha \cdot \nabla J$$

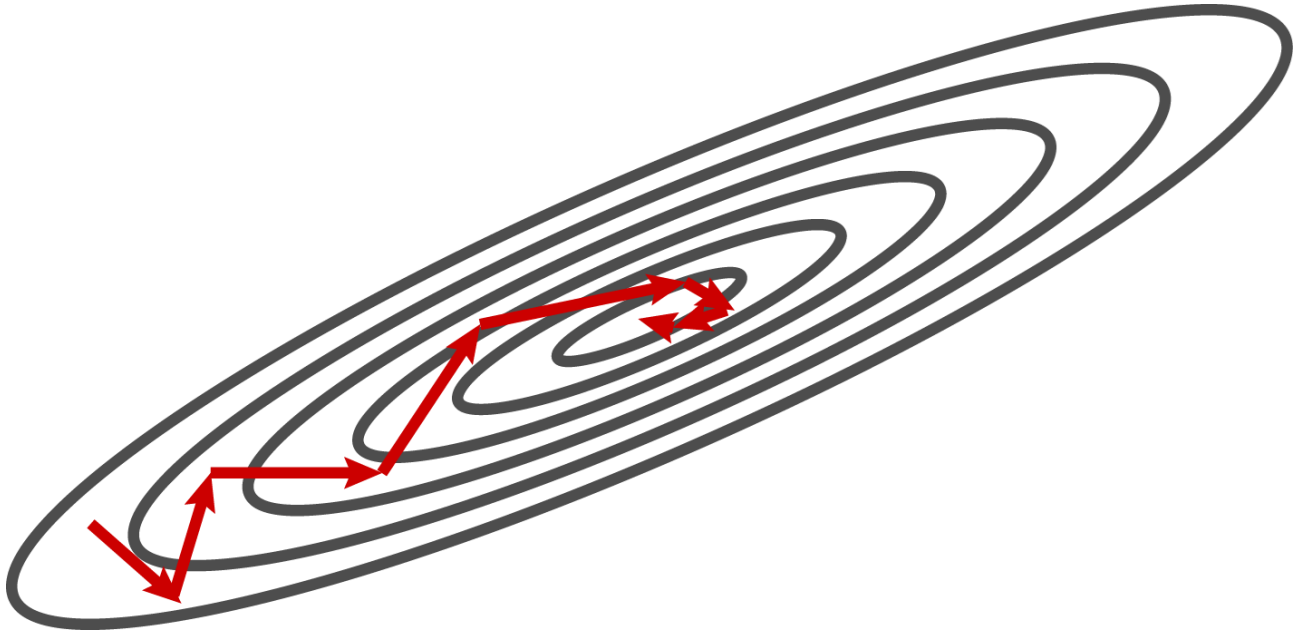
$$W := W - v_t$$

- Here, η is referred to as the “momentum”. It is generally given a value <1

Gradient Descent vs Momentum



Gradient Descent vs Momentum



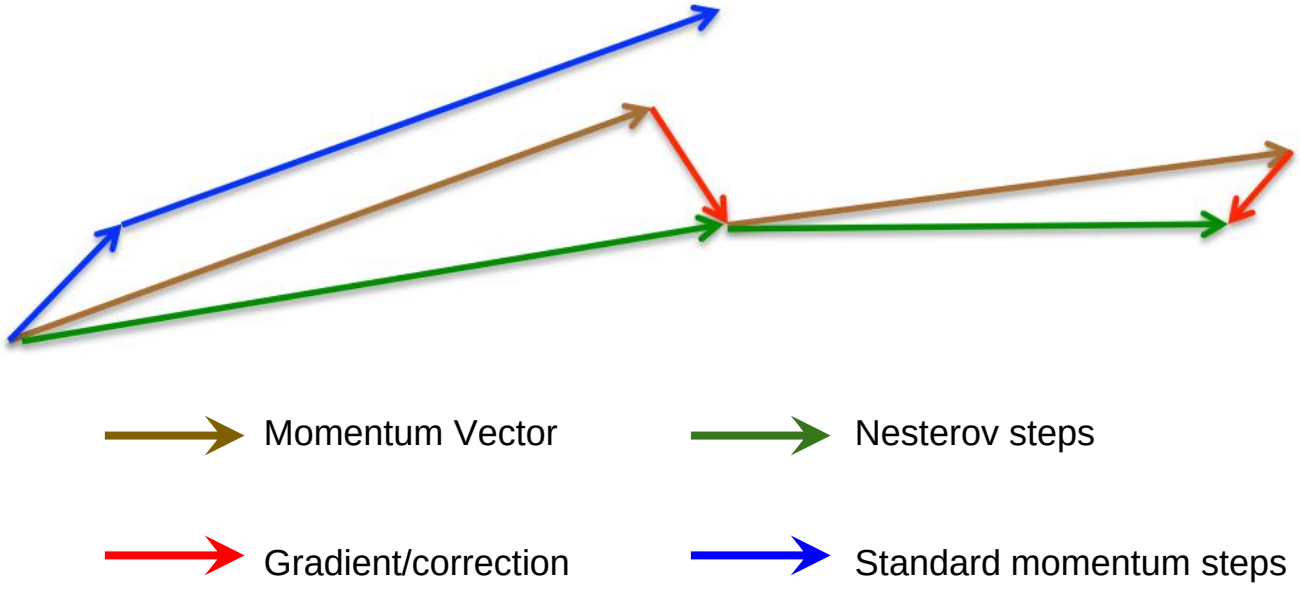
Nesterov Momentum

- Idea: Control “overshooting” by looking ahead.
- Apply gradient only to the “non-momentum” component.

$$v_t = \eta \cdot v_{t-1} - \alpha \cdot \nabla(J - \eta \cdot v_{t-1})$$

$$\mathrm{W} := W - v_t$$

Nesterov Momentum



AdaGrad

- Idea: scale the update for each weight separately.
- Update frequently-updated weights less
- Keep running sum of previous updates
- Divide new updates by factor of previous sum

$$W := W - \frac{\eta}{\sqrt{G_t} + \epsilon} \odot \nabla J$$

RMSProp

- Quite similar to AdaGrad.
- Rather than using the sum of previous gradients, decay older gradients more than more recent ones.
- More adaptive to recent updates

Adam

- Idea: use both first-order and second-order change information and decay both over time.

$$\mathrm{m_t} = \beta_1 \mathrm{m_{t-1}} + (1-\beta_1)\nabla \mathrm{J} \qquad \mathrm{v_t} = \beta_2 \mathrm{v_{t-1}} + (1-\beta_2)\nabla \mathrm{J}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1-\beta_1^t}$$

$$W := W - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \odot \hat{m}_t$$

Which one should I use?!

- RMSProp and Adam seem to be quite popular now.
- Difficult to predict in advance which will be best for a particular problem.
- Still an active area of inquiry.


intel®
Software

Introduction to Convolutional Neural Networks

Legal Notices and Disclaimers

This presentation is for informational purposes only. INTEL MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS SUMMARY.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. Check with your system manufacturer or retailer or learn more at intel.com.

This sample source code is released under the Intel Sample Source Code License Agreement.

Motivation – Image Data

- So far, the structure of our neural network treats all inputs interchangeably.
- No relationships between the individual inputs
- Just an ordered set of variables

- We want to incorporate domain knowledge into the architecture of a Neural Network.

Motivation

- Image data has important structures, such as;

- ”Topology” of pixels
- Translation invariance
- Issues of lighting and contrast
- Knowledge of human visual system
- Nearby pixels tend to have similar values
- Edges and shapes
- Scale Invariance – objects may appear at different sizes in the image.

Motivation – Image Data

- Fully connected would require a vast number of parameters
- MNIST images are small (32 x 32 pixels) and in grayscale
- Color images are more typically at least (200 x 200) pixels x 3 color channels (RGB) = 120,000 values.
- A single fully connected layer would require $(200x200x3)^2$ = 14,400,000,000 weights!
- Variance (in terms of bias-variance) would be too high
- So we introduce “bias” by structuring the network to look for certain kinds of patterns

Motivation

- Features need to be “built up”
- Edges -> shapes -> relations between shapes
- Textures

- Cat = two eyes in certain relation to one another + cat fur texture.
- Eyes = dark circle (pupil) inside another circle.
- Circle = particular combination of edge detectors.
- Fur = edges in certain pattern.

Kernels

- A *kernel* is a grid of weights “overlaid” on image, centered on one pixel
- Each weight multiplied with pixel underneath it
- Output over the centered pixel is $\sum_{p=1}^{P} W_p \cdot pixel_p$
- Used for traditional image processing techniques:
  - Blur
  - Sharpen
  - Edge detection
  - Emboss

Kernel: 3x3 Example

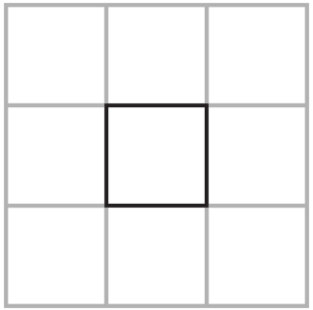
Input

3	2	1
1	2	3
1	1	1

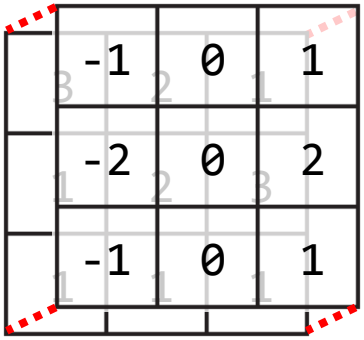
Kernel

-1	0	1
-2	0	2
-1	0	1

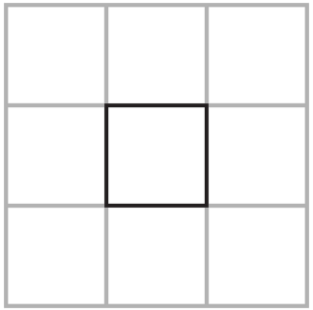
Output



Kernel: 3x3 Example



Output



Kernel: 3x3 Example

Input

3	2	1
1	2	3
1	1	1

Kernel

-1	0	1
-2	0	2
-1	0	1

Output

	2	

$$= (3 \cdot -1) + (2 \cdot 0) + (1 \cdot 1)$$
$$+(1 \cdot -2) + (2 \cdot 0) + (3 \cdot 2)$$
$$+(1 \cdot -1) + (1 \cdot 0) + (1 \cdot 1)$$

$$= -3 + 1 - 2 + 6 - 1 + 1 = 2$$

Kernels as Feature Detectors

Can think of kernels as a ”local feature detectors”

Vertical Line Detector

-1	1	-1
-1	1	-1
-1	1	-1

Horizontal Line Detector

-1	-1	-1
1	1	1
-1	-1	-1

Corner Detector

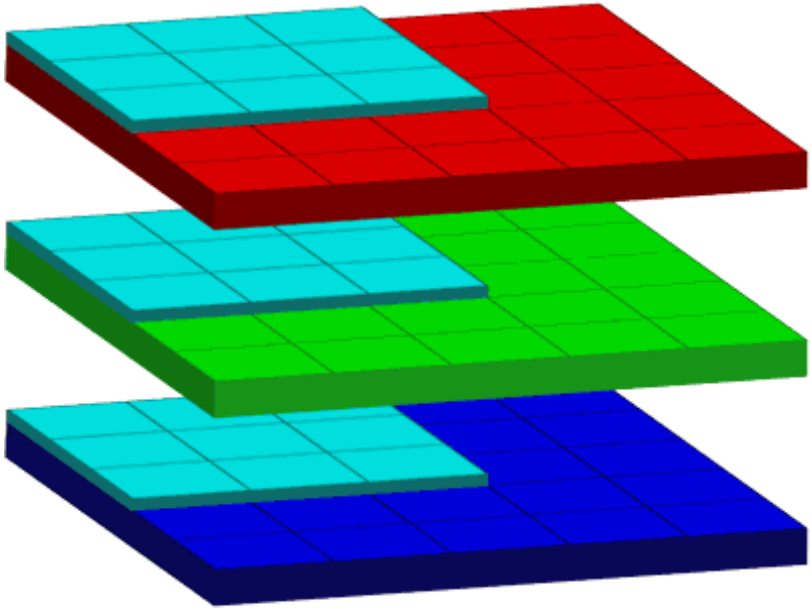
-1	-1	-1
-1	1	1
-1	1	1

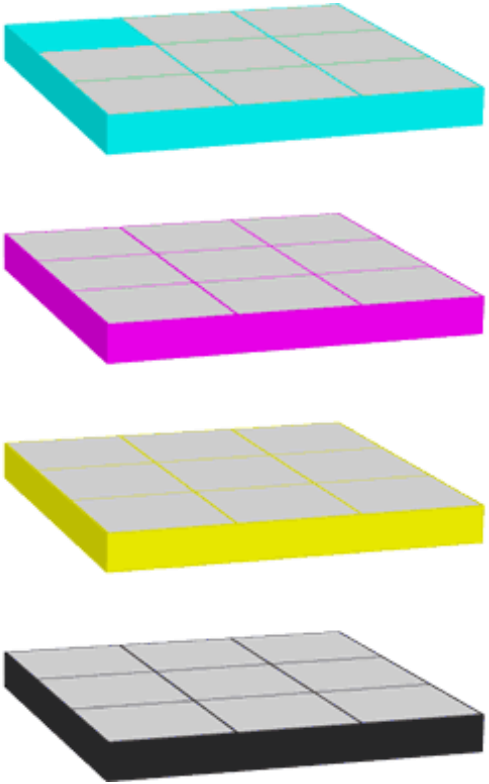
Convolutional Neural Nets

Primary Ideas behind Convolutional Neural Networks:

- Let the Neural Network learn which kernels are most useful
- Use same set of kernels across entire image (translation invariance)
- Reduces number of parameters and “variance” (from bias-variance point of view)

Convolutions



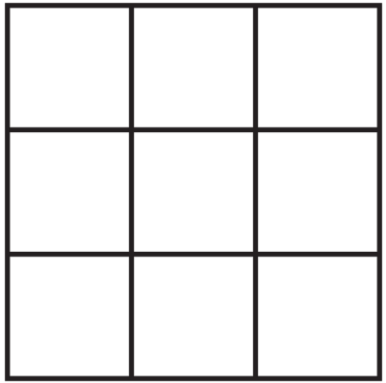


Convolution Settings – Grid Size

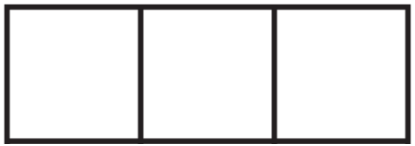
Grid Size (Height and Width):

- The number of pixels a kernel “sees” at once
- Typically use odd numbers so that there is a “center” pixel
- Kernel does not need to be square

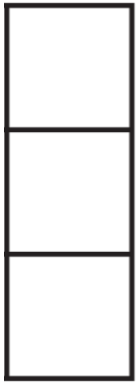
Height: 3, Width: 3



Height: 1, Width: 3



Height: 3, Width: 1

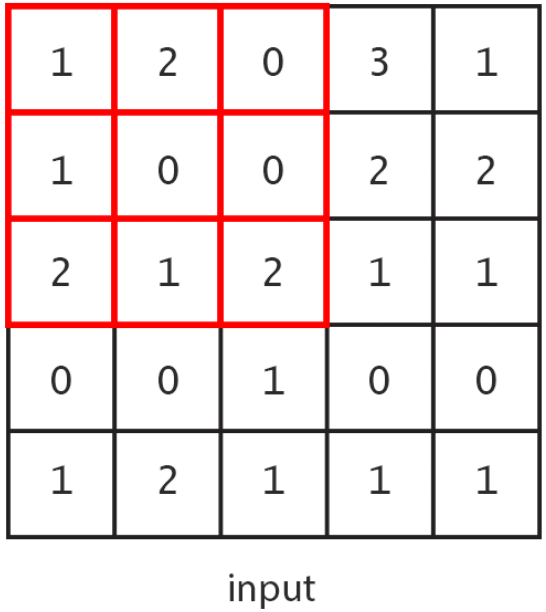


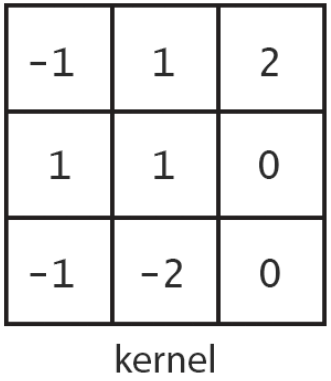
Convolution Settings - Padding

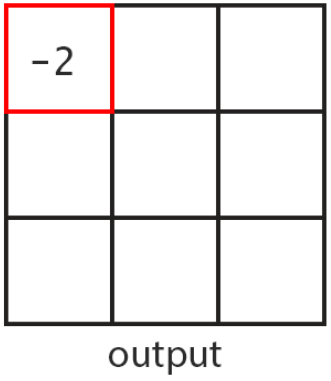
Padding

- Using Kernels directly, there will be an “edge effect”
- Pixels near the edge will not be used as “center pixels” since there are not enough surrounding pixels
- Padding adds extra pixels around the frame
- So every pixel of the original image will be a center pixel as the kernel moves across the image
- Added pixels are typically of value zero (zero-padding)

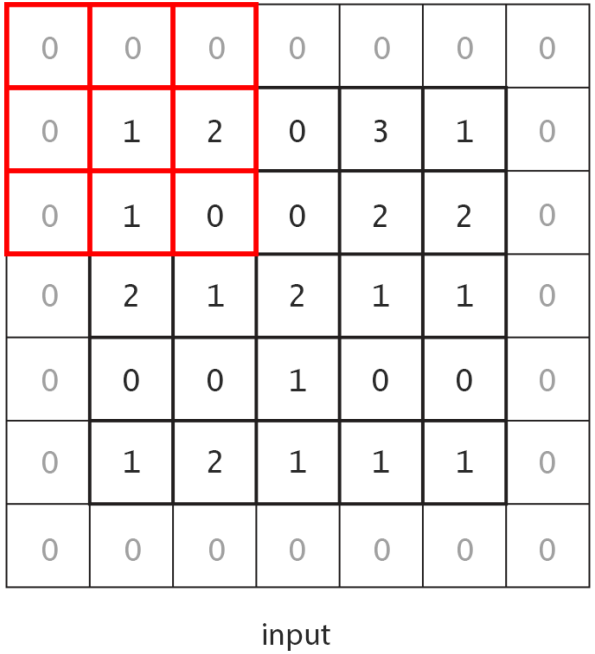
Without Padding

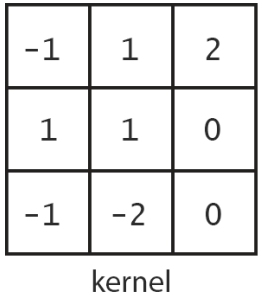


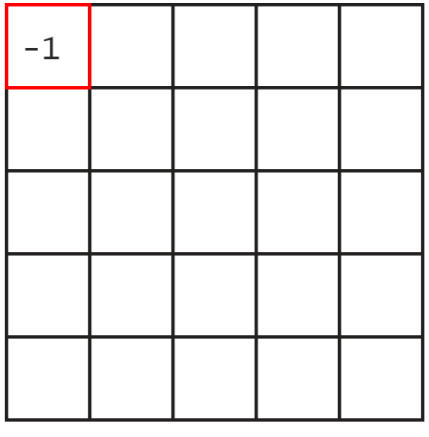




With Padding





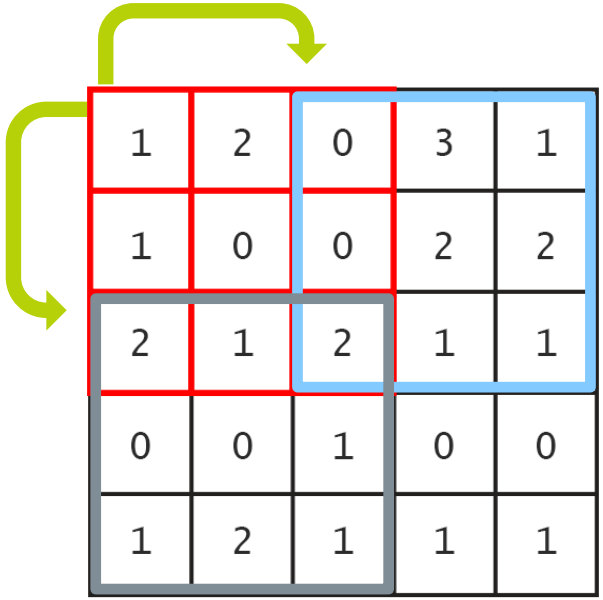


Convolution Settings

Stride

- The ”step size” as the kernel moves across the image
- Can be different for vertical and horizontal steps (but usually is the same value)
- When stride is greater than 1, it scales down the output dimension

Stride 2 Example – No Padding



input

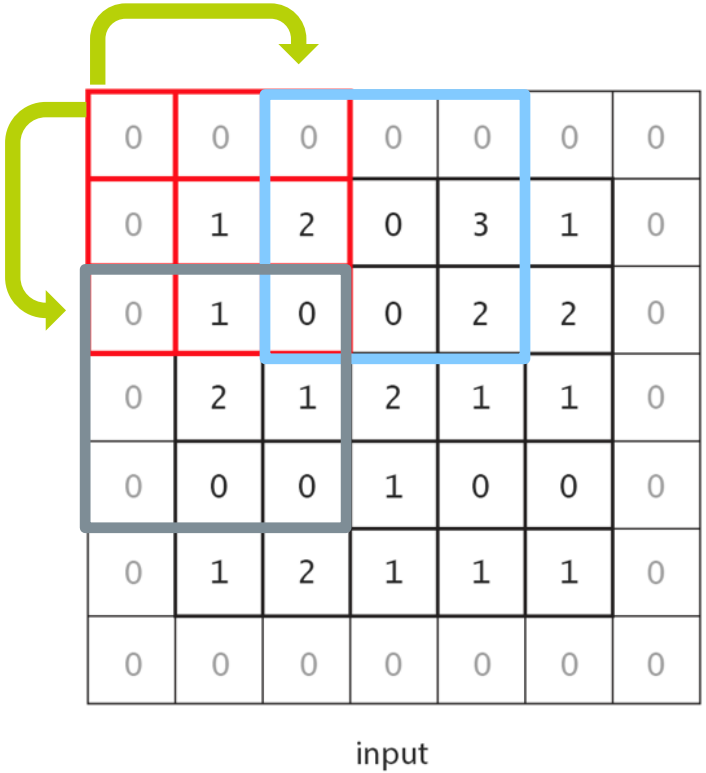
-1	1	2
1	1	0
-1	-2	0

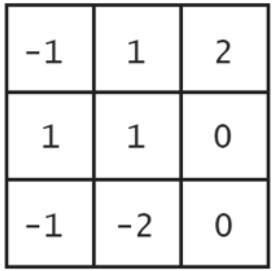
kernel

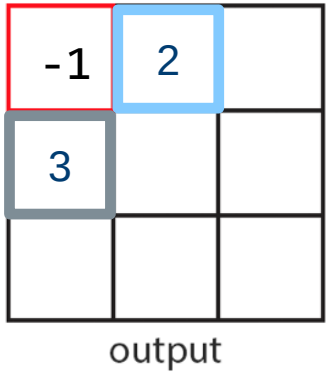
-2	3
0	

output

Stride 2 Example – With Padding







Convolutional Settings - Depth

- In images, we often have multiple numbers associated with each pixel location.
- These numbers are referred to as “channels”
  - RGB image – 3 channels
  - CMYK – 4 channels
- The number of channels is referred to as the “depth”
- So the kernel itself will have a “depth” the same size as the number of input channels
- Example: a 5x5 kernel on an RGB image
  - There will be 5x5x3 = 75 weights

Convolutional Settings - Depth

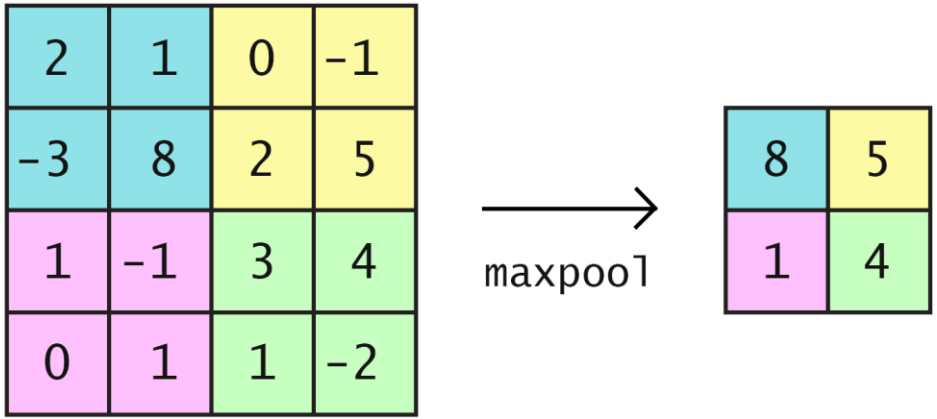
- The output from the layer will also have a depth
- The networks typically train many different kernels
- Each kernel outputs a single number at each pixel location
- So if there are 10 kernels in a layer, the output of that layer will have depth 10.

Pooling

- Idea: Reduce the image size by mapping a patch of pixels to a single value.
- Shrinks the dimensions of the image.
- Does not have parameters, though there are different types of pooling operations.

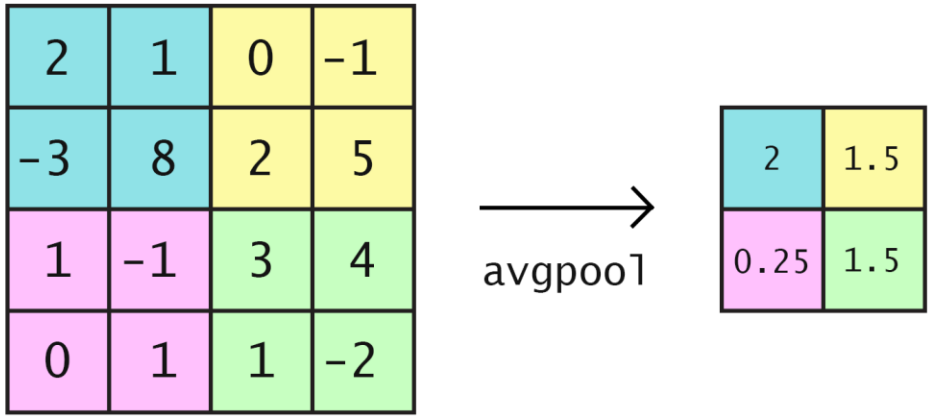
Pooling: Max-pool

- For each distinct patch, represent it by the maximum
- 2x2 maxpool shown below



Pooling: Average-pool

- For each distinct patch, represent it by the average
- 2x2 avgpool shown below.




intel
Software