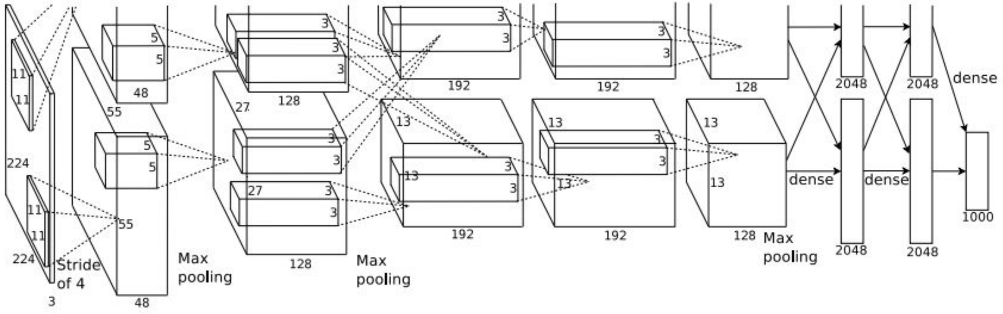
Lecture 2:
Deep Learning Fundamentals

Announcements

- A0 was released yesterday. Due next Tuesday, Sep 22.
    - Setup assignment for later homeworks.
- Project partner finding session this Friday, Sep 18.
- Office hours will start next week

Last time: key ingredients of deep learning success

Algorithms



Compute

Data

Today: Review of deep learning fundamentals

- Machine learning vs. deep learning framework

- Deep learning basics through a simple example
    - Defining a neural network architecture
    - Defining a loss function
    - Optimizing the loss function

- Model implementation using deep learning frameworks

- Design choices

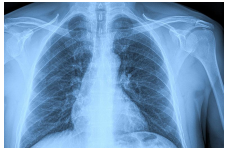
Machine learning framework

Data-driven learning of a mapping from input to output

Machine learning framework

Data-driven learning of a mapping from input to output

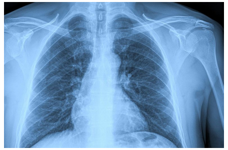
Traditional machine learning approaches

Input	**Feature extractor**	**Machine learning model**	**Output**
(e.g., )	(e.g., color and texture histograms)	(e.g., support vector machines and random forests)	(e.g., presence or not or disease)

Machine learning framework

Traditional machine learning

Input → **Feature extractor** → **Machine learning model** → **Output**

(e.g., )

(e.g., color and texture histograms)

(e.g., support vector machines and random forests)

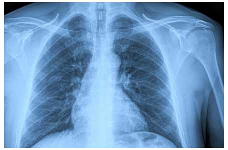
(e.g., presence or not or disease)

Q: What other features could be of interest in this X-ray? (Raise hand or type in chat box)

Deep learning (a type of machine learning)

Deep learning (a type of machine learning)

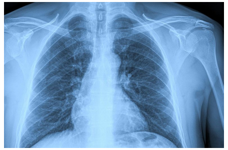
Traditional machine learning

Input		**Feature extractor**		**Machine learning model**		**Output**
(e.g., )	→	(e.g., color and texture histograms)	→	(e.g., support vector machines and random forests)	→	(e.g., presence or not or disease)

Deep learning (a type of machine learning)

Traditional machine learning

Input → **Feature extractor** → **Machine learning model** → **Output**

(e.g., )

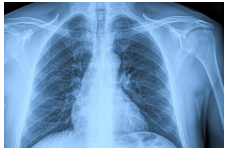
(e.g., color and texture histograms)

(e.g., support vector machines and random forests)

(e.g., presence or not or disease)

Deep learning

Input → **Deep Learning Model** → **Output**

(e.g., )

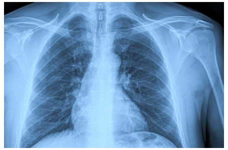
(e.g., convolutional and recurrent neural networks)

(e.g., presence or not or disease)

Deep learning (a type of machine learning)

Traditional machine learning

Input → **Feature extractor** → **Machine learning model** → **Output**

(e.g., )

(e.g., color and texture histograms)

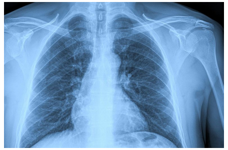
(e.g., support vector machines and random forests)

(e.g., presence or not or disease)

Deep learning

Directly learns what are useful (and better!) features from the training data

Input → **Deep Learning Model** → **Output**

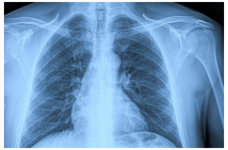
(e.g., )

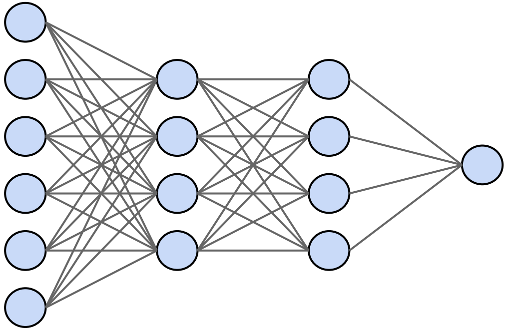
(e.g., convolutional and recurrent neural networks)

(e.g., presence or not or disease)

How do deep learning models perform feature extraction?

Input

(e.g., )

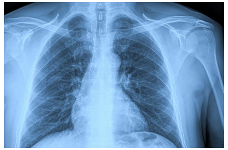


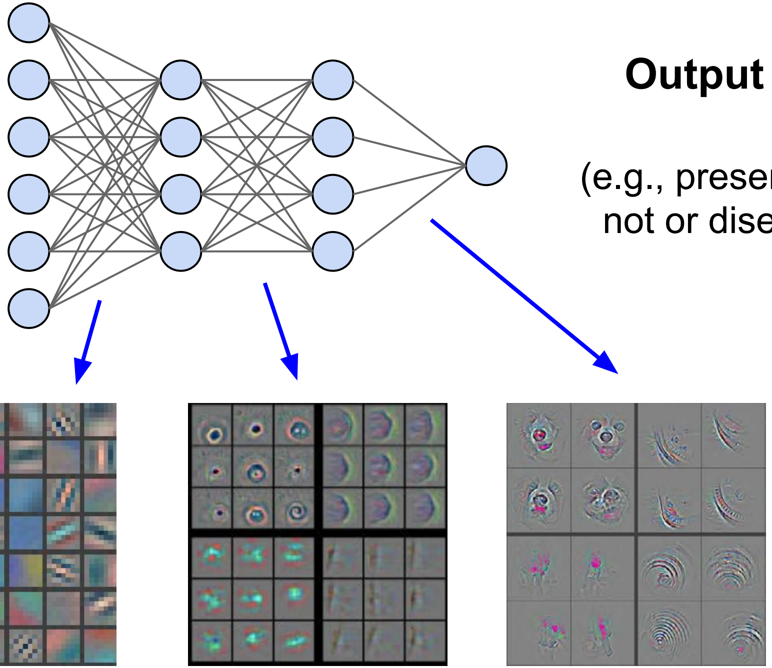
Output

(e.g., presence or not or disease)

How do deep learning models perform feature extraction?

Input

(e.g., )



Output

(e.g., presence or not or disease)

Hierarchical structure of neural networks allows compositional extraction of increasingly complex features

Low-level features

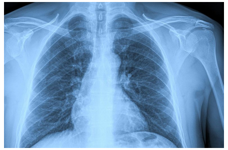
Mid-level features

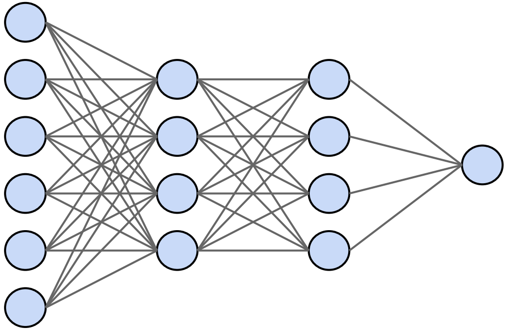
High-level features

Feature visualizations from Zeiler and Fergus 2013

Topics we will cover

Input

(e.g., )



Output

(e.g., presence or not or disease)

1. Preparing data for deep learning
2. Neural network models
3. Training neural networks
4. Evaluating models

First (today): deep learning basics through a simple example

Let us consider the task of **regression**: predicting a single real-valued output from input data

Model input: data vector $x = [x_1, x_2, ..., x_N]$

Model output: prediction (single number) $\hat{y}$

First (today): deep learning basics through a simple example

Let us consider the task of **regression**: predicting a single real-valued output from input data

Model input: data vector $x = [x_1, x_2, ..., x_N]$

Model output: prediction (single number) $\hat{y}$

Example: predicting hospital length-of-stay from clinical variables in the electronic health record

$x =$ [age, weight, …, temperature, oxygen saturation]

$\hat{y} =$ length-of-stay (days)

Example: predicting expression level of a target gene from the expression levels of N landmark genes

$x \in \mathcal{R}^N =$ expression levels of N landmark genes

$\hat{y} =$ expression level of target gene

Regression tasks

Breakout session:

1x 5-minute breakout (~4 students each)

- Introduce yourself!
- What other regression tasks are there in the medical field?
- What should be the inputs?
- What should be the output?
- What are the key features to extract?

Defining a neural network architecture

Our first architecture: a single-layer, fully connected neural network

Defining a neural network architecture

Our first architecture: a single-layer, fully connected neural network

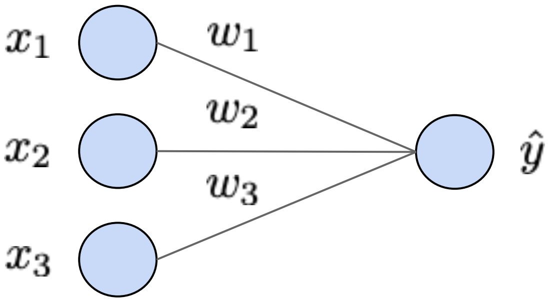
all inputs of a layer are connected to all outputs of a layer

Defining a neural network architecture

Our first architecture: a single-layer, fully connected neural network

For simplicity, use a 3-dimensional input (N = 3)

all inputs of a layer are connected to all outputs of a layer



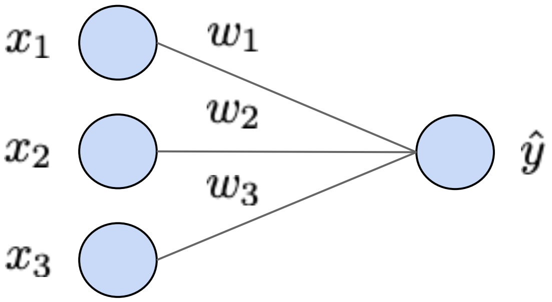
Output: $$\begin{aligned}\hat{y} &= w_1x_1 + w_2x_2 + w_3x_3 + b \\ &= w^Tx + b\end{aligned}$$

Defining a neural network architecture

Our first architecture: a single-layer, fully connected neural network

For simplicity, use a 3-dimensional input (N = 3)

all inputs of a layer are connected to all outputs of a layer



Output:
$$\begin{aligned}\hat{y} &= w_1x_1 + w_2x_2 + w_3x_3 + b \\ &= w^Tx + b\end{aligned}$$

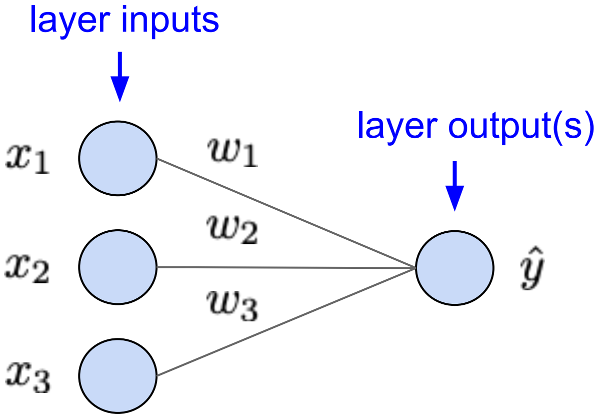
bias term (allows constant shift)

Defining a neural network architecture

Our first architecture: a single-layer, fully connected neural network

For simplicity, use a 3-dimensional input (N = 3)

all inputs of a layer are connected to all outputs of a layer



Output: $\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$

$= w^T x + b$

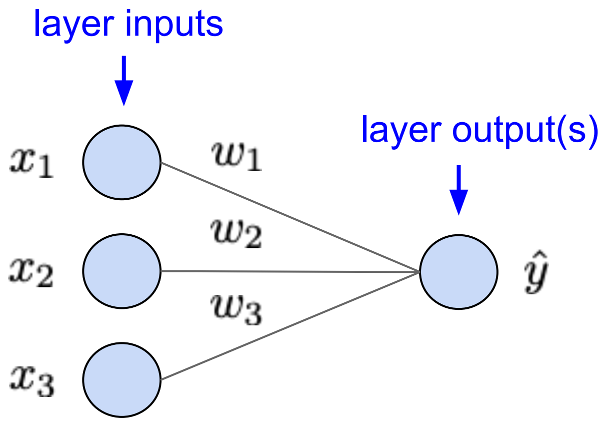
bias term (allows constant shift)

Defining a neural network architecture

Our first architecture: a single-layer, fully connected neural network

For simplicity, use a 3-dimensional input (N = 3)

all inputs of a layer are connected to all outputs of a layer



Output:
$$\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$
$$= w^T x + b$$

bias term (allows constant shift)

Neural network parameters:

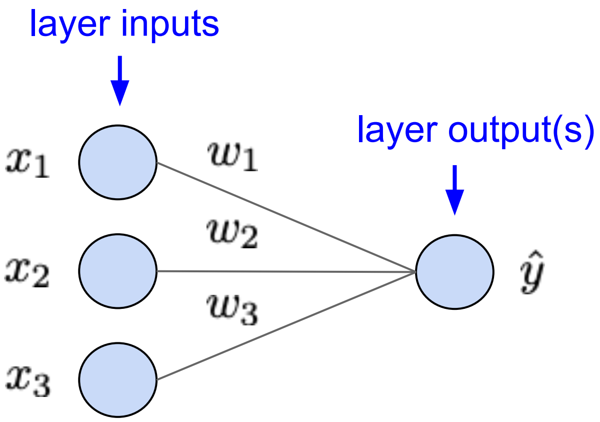
$$W = \{[w_1, w_2, w_3], b\}$$

Defining a neural network architecture

Our first architecture: a single-layer, fully connected neural network

For simplicity, use a 3-dimensional input (N = 3)

all inputs of a layer are connected to all outputs of a layer



Output:

$$\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$
$$= w^T x + b$$

bias term (allows constant shift)

Neural network parameters:

$$W = \{[w_1, w_2, w_3], b\}$$

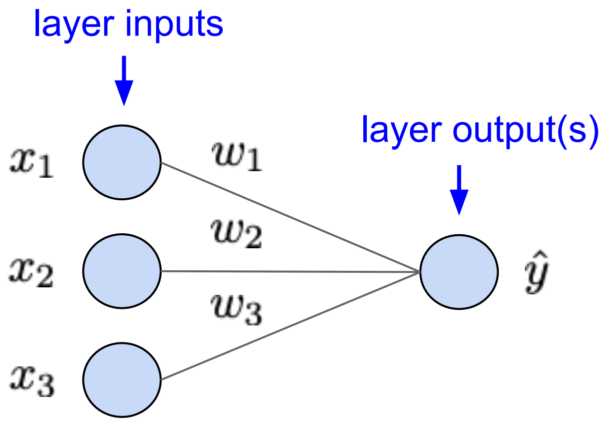
layer “weights” layer bias

Defining a neural network architecture

Our first architecture: a single-layer, fully connected neural network

For simplicity, use a 3-dimensional input (N = 3)

all inputs of a layer are connected to all outputs of a layer



Output:
$$\hat{y} = w_1x_1 + w_2x_2 + w_3x_3 + b$$
$$= w^Tx + b$$

bias term (allows constant shift)

Neural network parameters:

$$W = \{[w_1, w_2, w_3], b\}$$

layer "weights"

layer bias

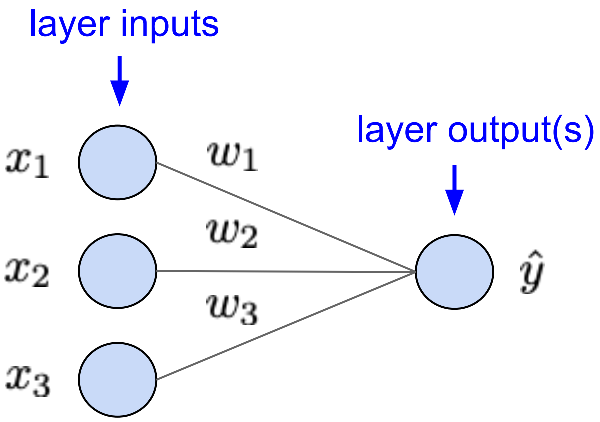
Often refer to all parameters together as just "weights". Bias is implicitly assumed.

Defining a neural network architecture

Our first architecture: a single-layer, fully connected neural network

For simplicity, use a 3-dimensional input (N = 3)

all inputs of a layer are connected to all outputs of a layer



Output:

$$\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$
$$= w^T x + b$$

bias term (allows constant shift)

Neural network parameters:

$$W = \{[w_1, w_2, w_3], b\}$$

layer "weights"

layer bias

Often refer to all parameters together as just "weights". Bias is implicitly assumed.

Caveats of our first (simple) neural network architecture:

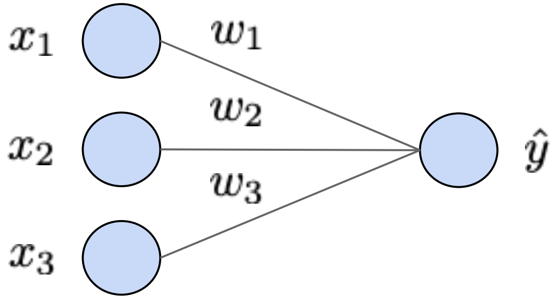
- Single layer still "shallow", not yet a "deep" neural network. Will see how to stack multiple layers.
- Also equivalent to a linear regression model! But useful base case for deep learning.

Defining a loss function

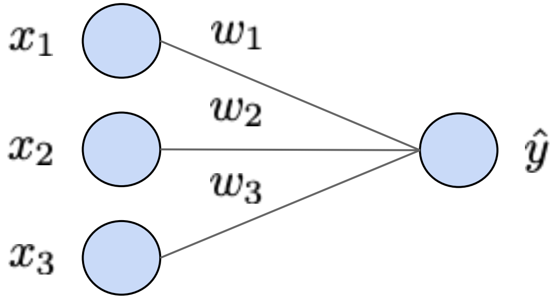
Output: $\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$
$= w^T x + b$

Neural network parameters:

$$W = \{[w_1, w_2, w_3], b\}$$



Defining a loss function



Output: $\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$
$= w^T x + b$

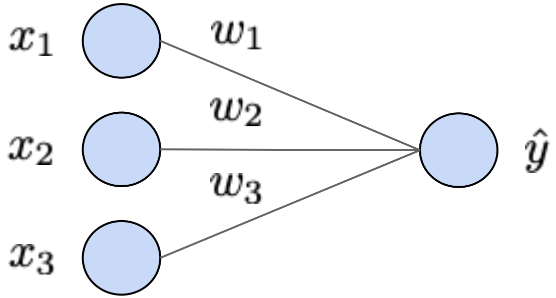
Neural network parameters:

$$W = \{[w_1, w_2, w_3], b\}$$

Loss functions are quantitative measures of how satisfactory the model predictions are (i.e., how "good" the model parameters are).

Defining a loss function

Output: $\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$
$= w^T x + b$



Neural network parameters:

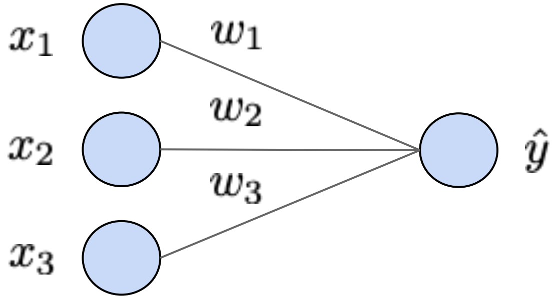
$$W = \{[w_1, w_2, w_3], b\}$$

Loss functions are quantitative measures of how satisfactory the model predictions are (i.e., how "good" the model parameters are).

We will use the mean square error (MSE) loss which is standard for regression.

Defining a loss function

Output: $\hat{y} = w_1x_1 + w_2x_2 + w_3x_3 + b$
$= w^Tx + b$



Neural network parameters:

$$W = \{[w_1, w_2, w_3], b\}$$

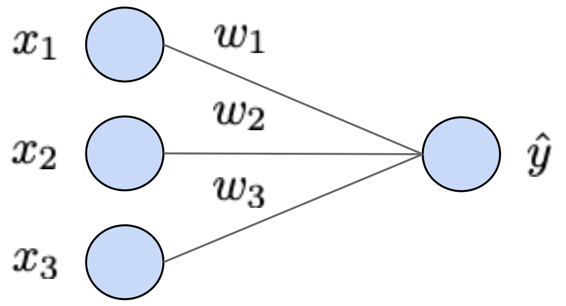
Loss functions are quantitative measures of how satisfactory the model predictions are (i.e., how "good" the model parameters are).

We will use the mean square error (MSE) loss which is standard for regression.

MSE loss for a single example x^i, when the prediction is $\hat{y}^i$ and the correct (ground truth) output is y^i

$$L^i(W) = (\hat{y}^i - y^i)^2$$

Defining a loss function



Output: $\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$
$$= w^T x + b$$

Neural network parameters:

$$W = \{[w_1, w_2, w_3], b\}$$

Loss functions are quantitative measures of how satisfactory the model predictions are (i.e., how “good” the model parameters are).

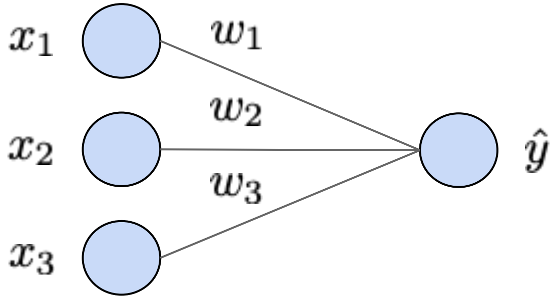
We will use the mean square error (MSE) loss which is standard for regression.

MSE loss for a single example x^i, when the prediction is $\hat{y}^i$ and the correct (ground truth) output is y^i

$$L^i(W) = (\hat{y}^i - y^i)^2$$ ⟵ the loss is small when the prediction is close to the ground truth

Defining a loss function

Output: $\hat{y} = w_1x_1 + w_2x_2 + w_3x_3 + b$
$= w^Tx + b$



Neural network parameters:

$$W = \{[w_1, w_2, w_3], b\}$$

Loss functions are quantitative measures of how satisfactory the model predictions are (i.e., how "good" the model parameters are).

We will use the mean square error (MSE) loss which is standard for regression.

MSE loss for a single example x^i, when the prediction is $\hat{y}^i$ and the correct (ground truth) output is y^i

$$L^i(W) = (\hat{y}^i - y^i)^2$$ ← the loss is small when the prediction is close to the ground truth

MSE loss over a set of examples $i = \{1, ..., M\}$: $L = \frac{1}{M}\sum_i L^i(W)$

Optimizing the loss function: gradient descent

Goal: find the “best” values of the model parameters that minimize the loss function

Optimizing the loss function: gradient descent

Goal: find the "best" values of the model parameters that minimize the loss function

The approach we will take: gradient descent

Optimizing the loss function: gradient descent

Goal: find the “best” values of the model parameters that minimize the loss function

The approach we will take: gradient descent

“Loss landscape”: the value of the loss function at every value of the model parameters

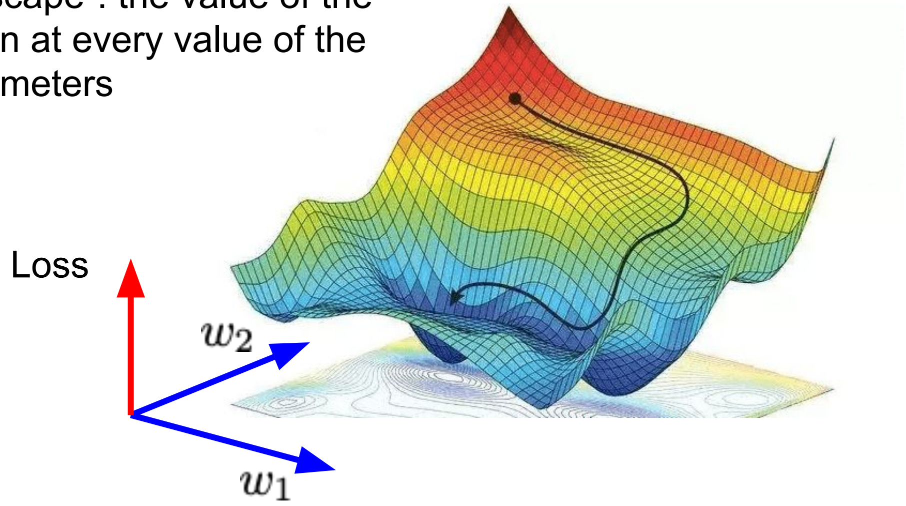


Figure credit: https://easyai.tech/wp-content/uploads/2019/01/tiduxiajiang-1.png

Optimizing the loss function: gradient descent

Goal: find the "best" values of the model parameters that minimize the loss function

The approach we will take: gradient descent

"Loss landscape": the value of the loss function at every value of the model parameters

Main idea: iteratively update the model parameters, to take steps in the local direction of steepest (negative) slope, i.e., the negative gradient

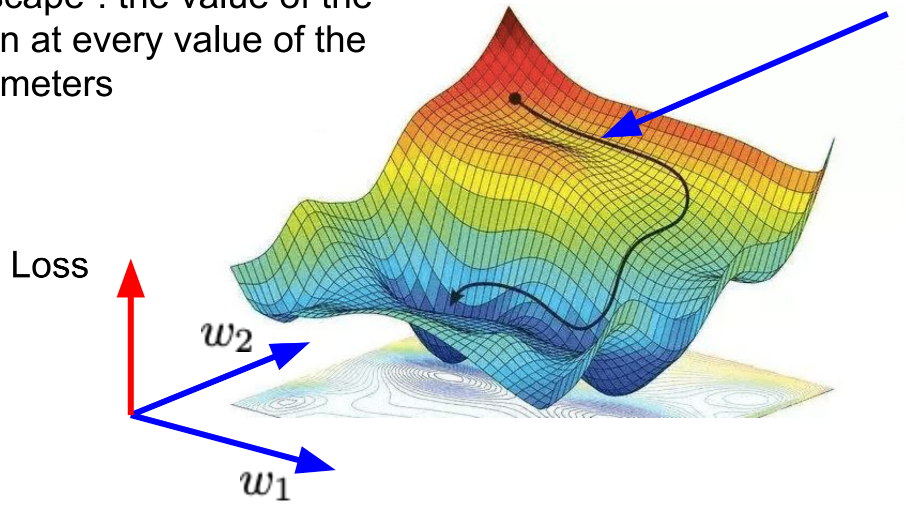


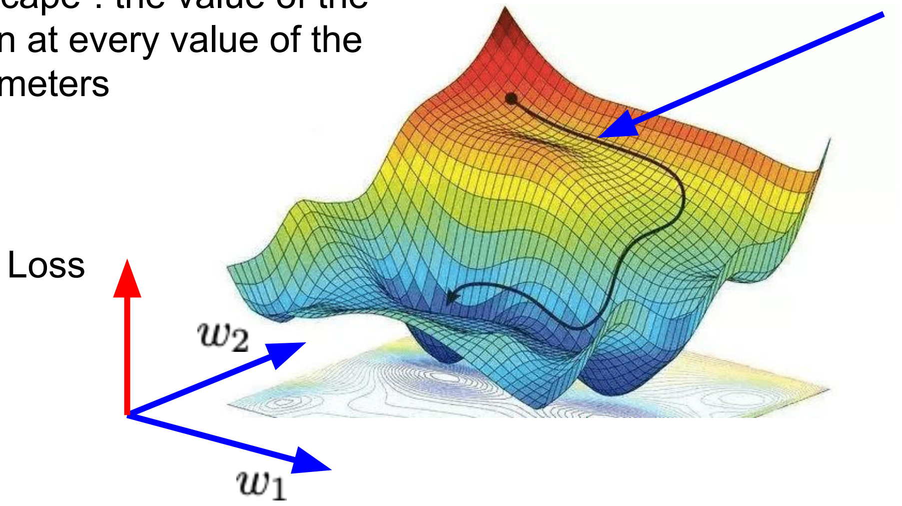
Figure credit: https://easyai.tech/wp-content/uploads/2019/01/tiduxiajiang-1.png

Optimizing the loss function: gradient descent

Goal: find the "best" values of the model parameters that minimize the loss function

The approach we will take: gradient descent

"Loss landscape": the value of the loss function at every value of the model parameters



Main idea: iteratively update the model parameters, to take steps in the local direction of steepest (negative) slope, i.e., the negative gradient

We will be able to use gradient descent to iteratively optimize the complex loss function landscapes corresponding to deep neural networks!

Figure credit: https://easyai.tech/wp-content/uploads/2019/01/tiduxiajiang-1.png

Review from calculus: derivatives and gradients

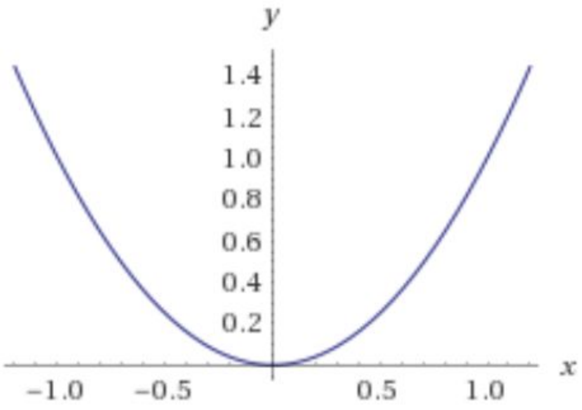
The **derivative** of a function is a measure of local slope.

Ex: $f(x) = x^2$ $\frac{\partial f}{\partial x} = 2x$

Review from calculus: derivatives and gradients

The **derivative** of a function is a measure of local slope.

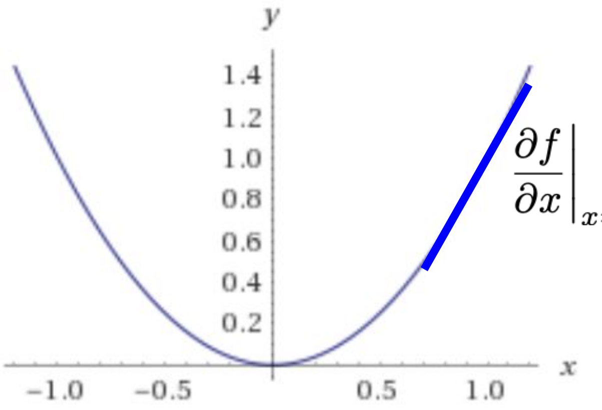
Ex: $f(x) = x^2$ $\qquad \frac{\partial f}{\partial x} = 2x$



Review from calculus: derivatives and gradients

The **derivative** of a function is a measure of local slope.

Ex: $f(x) = x^2$ $\qquad \dfrac{\partial f}{\partial x} = 2x$

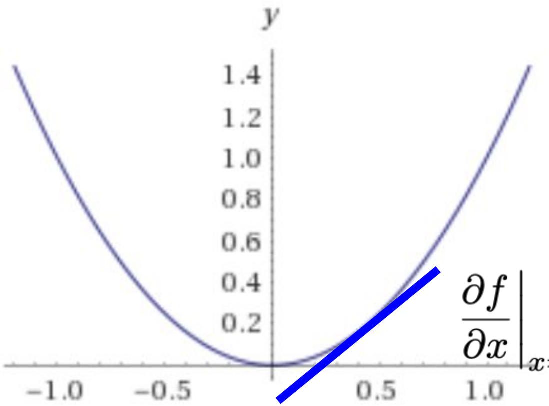


$$\left.\frac{\partial f}{\partial x}\right|_{x=1} = 2$$

Review from calculus: derivatives and gradients

The **derivative** of a function is a measure of local slope.

Ex: $f(x) = x^2$ $\qquad$ $\dfrac{\partial f}{\partial x} = 2x$

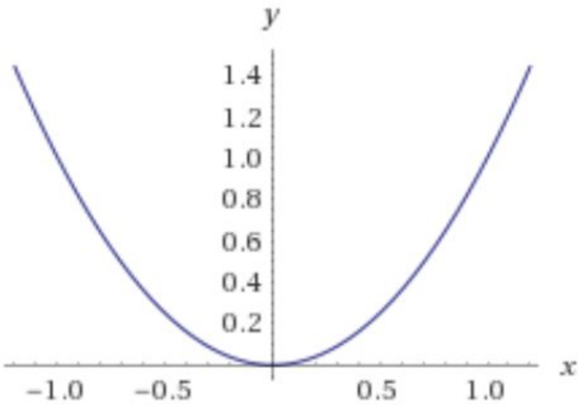


$$\left.\frac{\partial f}{\partial x}\right|_{x=0.5} = 1$$

Review from calculus: derivatives and gradients

The **derivative** of a function is a measure of local slope.

Ex: $f(x) = x^2$ $\quad\quad \frac{\partial f}{\partial x} = 2x$



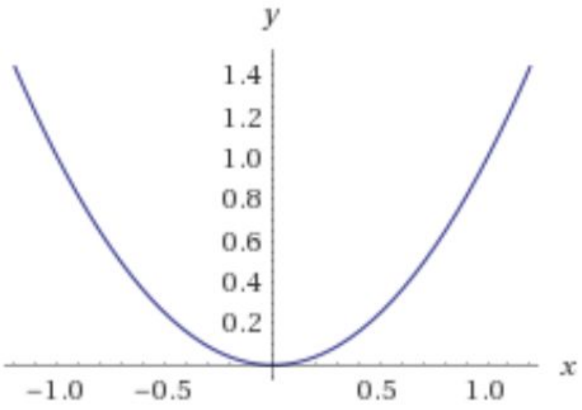
The **gradient** of a function of multiple variables is the vector of partial derivatives of the function with respect to each variable.

Ex: $f(x_1, x_2) = 3{x_1}^2 + {x_2}^2$ $\quad\quad \nabla f_x = [6x_1, 2x_2]$

Review from calculus: derivatives and gradients

The **derivative** of a function is a measure of local slope.

Ex: $f(x) = x^2$ $\qquad \dfrac{\partial f}{\partial x} = 2x$

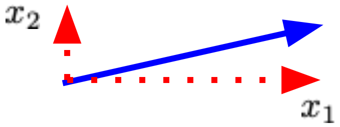


The **gradient** of a function of multiple variables is the vector of partial derivatives of the function with respect to each variable.

Ex: $f(x_1, x_2) = 3{x_1}^2 + {x_2}^2$ $\qquad \nabla f_x = [6x_1, 2x_2]$

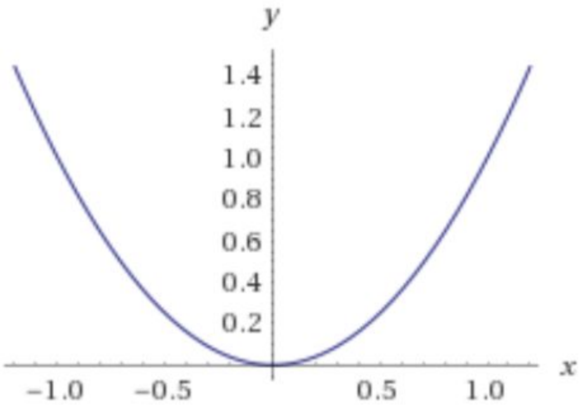
The gradient evaluated at a particular point is the direction of steepest ascent of the function.

$$\nabla f_x \Big|_{x_1=1, x_2=1} = [6, 2]$$



Review from calculus: derivatives and gradients

The **derivative** of a function is a measure of local slope.



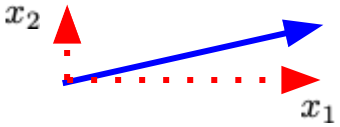
Ex: $f(x) = x^2$ $\qquad \dfrac{\partial f}{\partial x} = 2x$

The **gradient** of a function of multiple variables is the vector of partial derivatives of the function with respect to each variable.

Ex: $f(x_1, x_2) = 3{x_1}^2 + {x_2}^2$ $\qquad \nabla f_x = [6x_1, 2x_2]$

The gradient evaluated at a particular point is the direction of steepest ascent of the function.

$$\nabla f_x \Big|_{x_1=1, x_2=1} = [6, 2]$$



The negative of the gradient is the direction of steepest descent -> direction we want to move in the loss function landscape!

Gradient descent algorithm

Let the gradient of the loss function with respect to the model parameters w be:

$$\nabla L_W = \left[\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_K}\right]$$

Gradient descent algorithm

Let the gradient of the loss function with respect to the model parameters w be:

$$\nabla L_W = \left[\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_K}\right]$$

For ease of notation, rewrite parameter b as w_0 corresponding to $x_0 = 1$:

$\hat{y} = w_0 x_0 + w_1 x_1 + w_2 x_2 + w_3 x_3$

$W = \{[w_0, w_1, w_2, w_3]\}$

Gradient descent algorithm

Let the gradient of the loss function with respect to the model parameters w be:

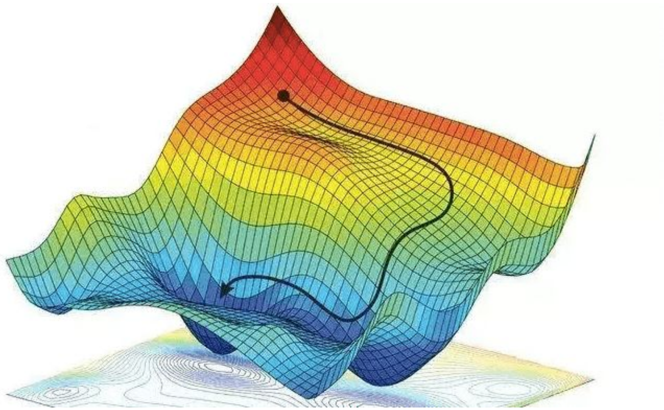
$$\nabla L_W = \left[\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_K}\right]$$

For ease of notation, rewrite parameter b as w_0 corresponding to $x_0 = 1$:

$$\hat{y} = w_0x_0 + w_1x_1 + w_2x_2 + w_3x_3$$
$$W = \{[w_0, w_1, w_2, w_3]\}$$

Then we can minimize the loss function by iteratively updating the model parameters ("taking steps") in the direction of the negative gradient, until convergence:

$$W := W - \alpha\nabla L_W$$



Gradient descent algorithm

Let the gradient of the loss function with respect to the model parameters w be:

$$\nabla L_W = \left[\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}, \ldots, \frac{\partial L}{\partial w_K}\right]$$

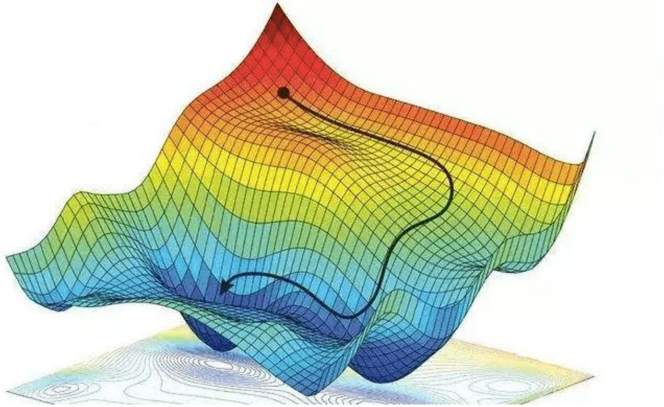
For ease of notation, rewrite parameter b as w_0 corresponding to $x_0 = 1$:

$$\hat{y} = w_0x_0 + w_1x_1 + w_2x_2 + w_3x_3$$
$$W = \{[w_0, w_1, w_2, w_3]\}$$

Then we can minimize the loss function by iteratively updating the model parameters ("taking steps") in the direction of the negative gradient, until convergence:

$$W := W - \alpha \nabla L_W$$

"Step size" hyperparameter (design choice) indicating how big of a step in the negative gradient direction we want to take at each update.
Too big -> may overshoot minima.
Too small -> optimization takes too long.



Gradient descent algorithm: in code

```
# initialize vector of weight parameters to random values
weights = random_init(weights_dimension)

while True:
    # evaluate the gradient of the loss function with respect to the weights
    weights_grad = evaluate_gradient(loss_fcn, data, weights)
    # update the weights in the direction of the negative gradient
    weights = weights - step_size * weights_grad
```

Stochastic gradient descent (SGD)

Evaluating gradient involves iterating over all data examples, which can be slow!

In practice, usually use stochastic gradient descent: **estimate gradient over a sample of data examples** (usually as many as can fit on GPU at one time, e.g. 32, 64, 128)

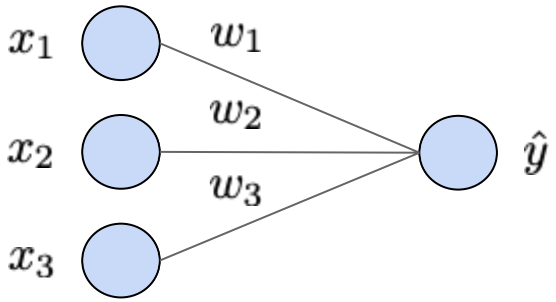
```
# initialize vector of weight parameters to random values
weights = random_init(weights_dimension)

while True:

    # sample a batch of data examples
    data_batch = sample_data(data, 128)

    # evaluate the gradient of the loss function with respect to the weights
    weights_grad = evaluate_gradient(loss_fcn, data_batch, weights)
    # update the weights in the direction of the negative gradient
    weights = weights - step_size * weights_grad
```

Optimizing the loss function: our example



Output: $$\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$$
$$= w^T x + b$$

Neural network parameters:

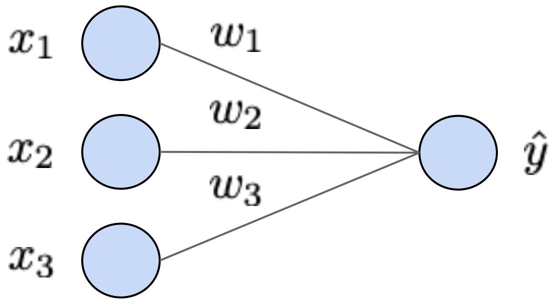
$$W = \{[w_1, w_2, w_3], b\}$$

Loss function:

Per-example: $L^i(W) = (\hat{y}^i - y^i)^2$

Over M examples: $L = \frac{1}{M} \sum_i L^i(W)$

Optimizing the loss function: our example



Output: $\hat{y} = w_1x_1 + w_2x_2 + w_3x_3 + b$

$= w^T x + b$

Neural network parameters:

$$W = \{[w_1, w_2, w_3], b\}$$

Loss function:

Per-example: $L^i(W) = (\hat{y}^i - y^i)^2$

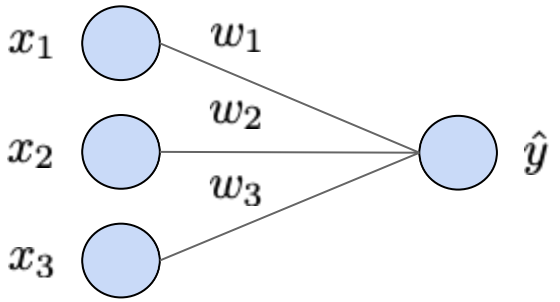
Over M examples: $L = \frac{1}{M}\sum_i L^i(W)$

Gradient of loss w.r.t. weights:

Partial derivative of loss w.r.t. kth weight:

$$\frac{\partial L^i}{\partial w_k} = \frac{\partial L^i}{\partial \hat{y}^i}\frac{\partial \hat{y}^i}{\partial w_k} = 2(\hat{y}^i - y^i)x_k^i$$

Optimizing the loss function: our example



Output: $\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$

$= w^T x + b$

Neural network parameters:

$$W = \{[w_1, w_2, w_3], b\}$$

Loss function:

Per-example: $L^i(W) = (\hat{y}^i - y^i)^2$

Over M examples: $L = \frac{1}{M} \sum_i L^i(W)$

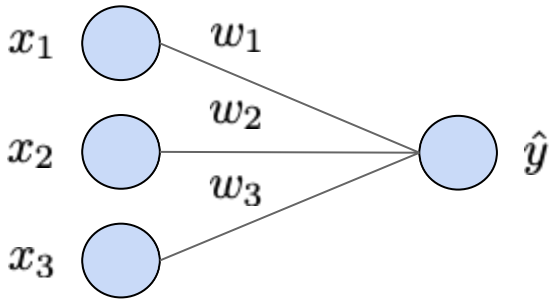
Gradient of loss w.r.t. weights:

Partial derivative of loss w.r.t. kth weight:

Chain rule

$$\frac{\partial L^i}{\partial w_k} = \frac{\partial L^i}{\partial \hat{y}^i} \frac{\partial \hat{y}^i}{\partial w_k} = 2(\hat{y}^i - y^i) x_k^i$$

Optimizing the loss function: our example



Output: $$\hat{y} = w_1x_1 + w_2x_2 + w_3x_3 + b$$
$$= w^Tx + b$$

Neural network parameters:

$$W = \{[w_1, w_2, w_3], b\}$$

Loss function:

Per-example: $L^i(W) = (\hat{y}^i - y^i)^2$

Over M examples: $L = \frac{1}{M}\sum_i L^i(W)$

Gradient of loss w.r.t. weights:

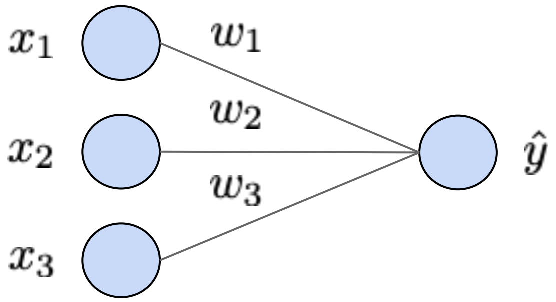
Partial derivative of loss w.r.t. kth weight:

$$\frac{\partial L^i}{\partial w_k} = \frac{\partial L^i}{\partial \hat{y}^i}\frac{\partial \hat{y}^i}{\partial w_k} = 2(\hat{y}^i - y^i)x_k^i$$

Over M examples

$$\frac{\partial L}{\partial w_k} = \frac{1}{M}\sum_i \frac{\partial L^i}{\partial w_k} = \frac{1}{M}\sum_i 2(\hat{y}^i - y^i)x_k^i$$

Optimizing the loss function: our example



Output: $\hat{y} = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$
$= w^T x + b$

Neural network parameters:

$$W = \{[w_1, w_2, w_3], b\}$$

Loss function:

Per-example: $L^i(W) = (\hat{y}^i - y^i)^2$

Over M examples: $L = \frac{1}{M} \sum_i L^i(W)$

Gradient of loss w.r.t. weights:

Partial derivative of loss w.r.t. kth weight:

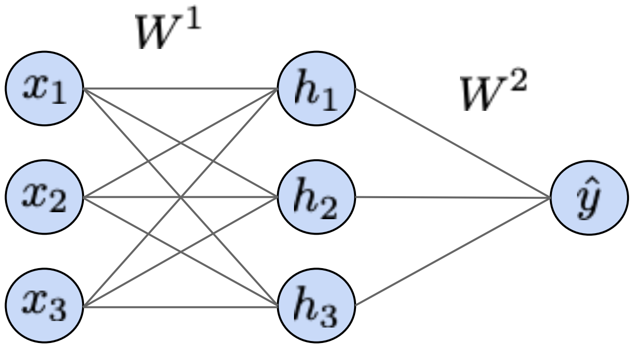
$$\frac{\partial L^i}{\partial w_k} = \frac{\partial L^i}{\partial \hat{y}^i} \frac{\partial \hat{y}^i}{\partial w_k} = 2(\hat{y}^i - y^i) x_k^i$$

$$\frac{\partial L}{\partial w_k} = \frac{1}{M} \sum_i \frac{\partial L^i}{\partial w_k} = \frac{1}{M} \sum_i 2(\hat{y}^i - y^i) x_k^i$$

Full gradient expression:

$$\nabla L_W = \left[\frac{\partial L}{w_0}, \ldots, \frac{\partial L}{w_3}\right] = \boxed{\frac{1}{M} \sum_i 2(\hat{y}^i - y^i) x^i}$$

Now: a two-layer fully-connected neural network



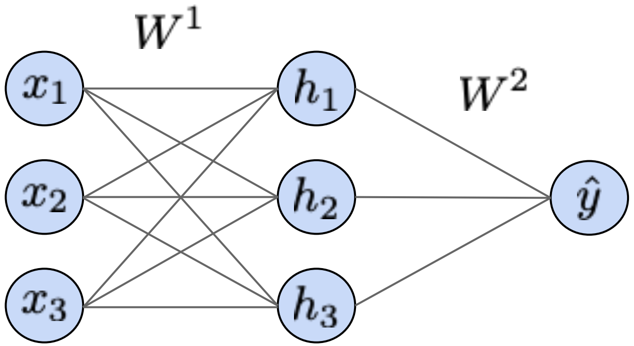
Output: $h = \sigma(W^1 x + b^1)$

$\hat{y} = W^2 h + b^2$

$$W^1 = \begin{bmatrix} w^1_{11} & w^1_{12} & w^1_{13} \\ w^1_{21} & w^1_{22} & w^1_{23} \\ w^1_{31} & w^1_{32} & w^1_{33} \end{bmatrix} \quad b^1 = \begin{bmatrix} b^1_1 \\ b^1_2 \\ b^1_3 \end{bmatrix}$$

$$W^2 = \begin{bmatrix} w^2_{11} & w^2_{12} & w^2_{13} \end{bmatrix} \quad b^2 = \begin{bmatrix} b^2_1 \end{bmatrix}$$

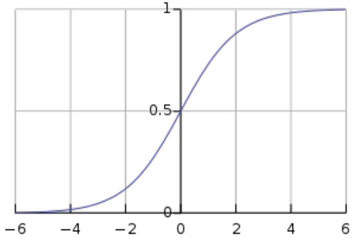
Now: a two-layer fully-connected neural network



Output: $h = \sigma(W^1 x + b^1)$

$\hat{y} = W^2 h + b^2$

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

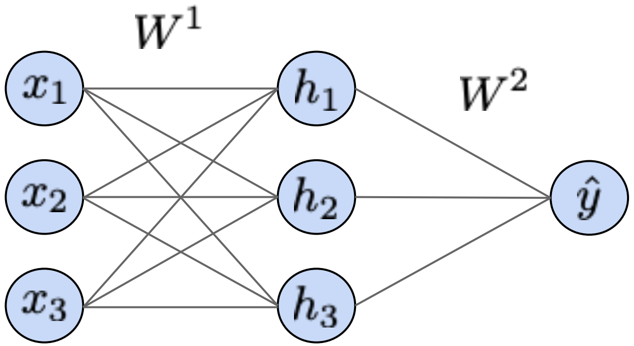


Sigmoid "activation function"

$$W^1 = \begin{bmatrix} w^1_{11} & w^1_{12} & w^1_{13} \\ w^1_{21} & w^1_{22} & w^1_{23} \\ w^1_{31} & w^1_{32} & w^1_{33} \end{bmatrix} \quad b^1 = \begin{bmatrix} b^1_1 \\ b^1_2 \\ b^1_3 \end{bmatrix}$$

$$W^2 = \begin{bmatrix} w^2_{11} & w^2_{12} & w^2_{13} \end{bmatrix} \quad b^2 = \begin{bmatrix} b^2_1 \end{bmatrix}$$

Now: a two-layer fully-connected neural network



Output: $h = \sigma(W^1 x + b^1)$

$$\hat{y} = W^2 h + b^2$$

Full function expression:

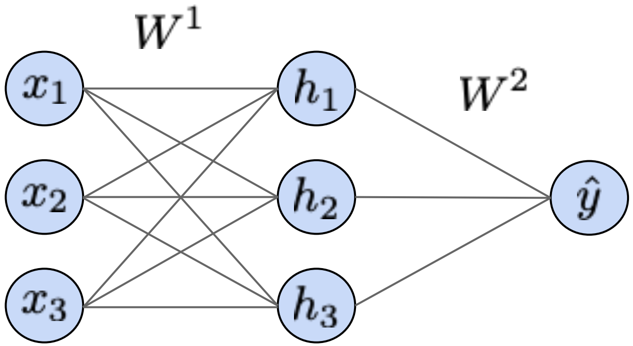
$$\hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2$$

$$W^1 = \begin{bmatrix} w^1_{11} & w^1_{12} & w^1_{13} \\ w^1_{21} & w^1_{22} & w^1_{23} \\ w^1_{31} & w^1_{32} & w^1_{33} \end{bmatrix} \quad b^1 = \begin{bmatrix} b^1_1 \\ b^1_2 \\ b^1_3 \end{bmatrix}$$

$$W^2 = \begin{bmatrix} w^2_{11} & w^2_{12} & w^2_{13} \end{bmatrix} \quad b^2 = \begin{bmatrix} b^2_1 \end{bmatrix}$$

Now: a two-layer fully-connected neural network

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

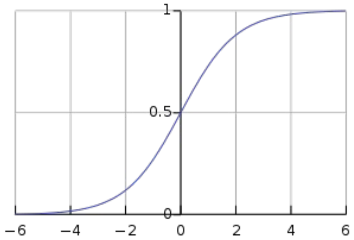


Output: $h = \sigma(W^1 x + b^1)$

$$\hat{y} = W^2 h + b^2$$

Full function expression:

$$\hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2$$



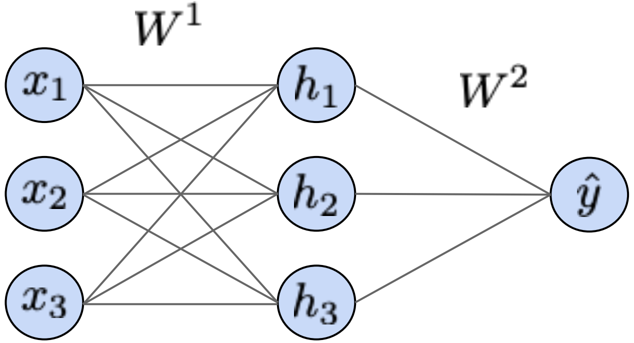
Sigmoid "activation function"

$$W^1 = \begin{bmatrix} w^1_{11} & w^1_{12} & w^1_{13} \\ w^1_{21} & w^1_{22} & w^1_{23} \\ w^1_{31} & w^1_{32} & w^1_{33} \end{bmatrix} \quad b^1 = \begin{bmatrix} b^1_1 \\ b^1_2 \\ b^1_3 \end{bmatrix}$$

$$W^2 = \begin{bmatrix} w^2_{11} & w^2_{12} & w^2_{13} \end{bmatrix} \quad b^2 = \begin{bmatrix} b^2_1 \end{bmatrix}$$

Activation functions
Introduce non-linearity into the model -- allowing it to represent highly complex functions.

Now: a two-layer fully-connected neural network



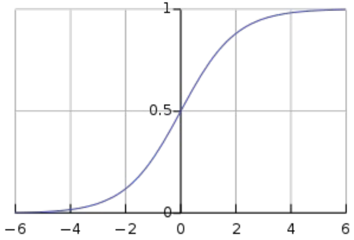
$$W^1 = \begin{bmatrix} w^1_{11} & w^1_{12} & w^1_{13} \\ w^1_{21} & w^1_{22} & w^1_{23} \\ w^1_{31} & w^1_{32} & w^1_{33} \end{bmatrix} \quad b^1 = \begin{bmatrix} b^1_1 \\ b^1_2 \\ b^1_3 \end{bmatrix}$$

$$W^2 = \begin{bmatrix} w^2_{11} & w^2_{12} & w^2_{13} \end{bmatrix} \quad b^2 = \begin{bmatrix} b^2_1 \end{bmatrix}$$

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

Output: $h = \sigma(W^1 x + b^1)$

$\hat{y} = W^2 h + b^2$



Sigmoid "activation function"

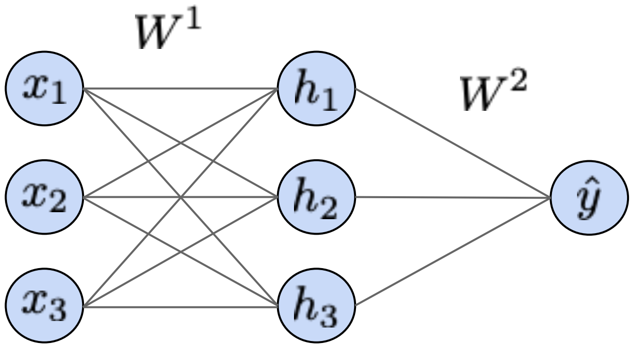
Full function expression:

$$\hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2$$

Activation functions introduce non-linearity into the model -- allowing it to represent highly complex functions.

A **fully-connected neural network** (also known as multi-layer perceptron) is a stack of [affine transformation + activation function] layers.

Now: a two-layer fully-connected neural network

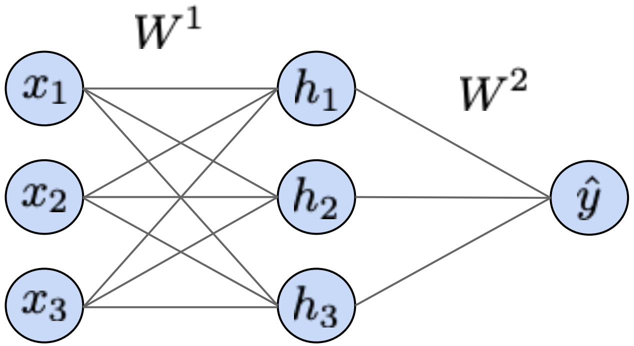


Output: $\hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2$

$$W^1 = \begin{bmatrix} w^1_{11} & w^1_{12} & w^1_{13} \\ w^1_{21} & w^1_{22} & w^1_{23} \\ w^1_{31} & w^1_{32} & w^1_{33} \end{bmatrix} \quad b^1 = \begin{bmatrix} b^1_1 \\ b^1_2 \\ b^1_3 \end{bmatrix}$$

$$W^2 = \begin{bmatrix} w^2_{11} & w^2_{12} & w^2_{13} \end{bmatrix} \quad b^2 = \begin{bmatrix} b^2_1 \end{bmatrix}$$

Now: a two-layer fully-connected neural network



Output: $\hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2$

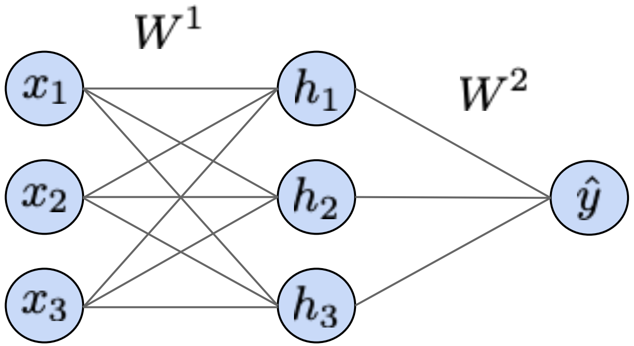
Neural network parameters:

$$W = \{W^1, b^1, W^2, b^2\}$$

$$W^1 = \begin{bmatrix} w^1_{11} & w^1_{12} & w^1_{13} \\ w^1_{21} & w^1_{22} & w^1_{23} \\ w^1_{31} & w^1_{32} & w^1_{33} \end{bmatrix} \quad b^1 = \begin{bmatrix} b^1_1 \\ b^1_2 \\ b^1_3 \end{bmatrix}$$

$$W^2 = \begin{bmatrix} w^2_{11} & w^2_{12} & w^2_{13} \end{bmatrix} \quad b^2 = \begin{bmatrix} b^2_1 \end{bmatrix}$$

Now: a two-layer fully-connected neural network



$$W^1 = \begin{bmatrix} w^1_{11} & w^1_{12} & w^1_{13} \\ w^1_{21} & w^1_{22} & w^1_{23} \\ w^1_{31} & w^1_{32} & w^1_{33} \end{bmatrix} \quad b^1 = \begin{bmatrix} b^1_1 \\ b^1_2 \\ b^1_3 \end{bmatrix}$$

$$W^2 = \begin{bmatrix} w^2_{11} & w^2_{12} & w^2_{13} \end{bmatrix} \quad b^2 = \begin{bmatrix} b^2_1 \end{bmatrix}$$

Output: $\hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2$

Neural network parameters:

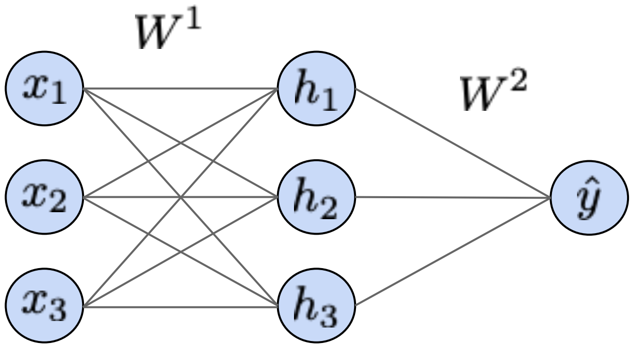
$$W = \{W^1, b^1, W^2, b^2\}$$

Loss function (regression loss, same as before)**:**

Per-example: $L^i(W) = (\hat{y}^i - y^i)^2$

Over M examples: $L = \frac{1}{M} \sum_i L^i(W)$

Now: a two-layer fully-connected neural network



$$W^1 = \begin{bmatrix} w^1_{11} & w^1_{12} & w^1_{13} \\ w^1_{21} & w^1_{22} & w^1_{23} \\ w^1_{31} & w^1_{32} & w^1_{33} \end{bmatrix} \quad b^1 = \begin{bmatrix} b^1_1 \\ b^1_2 \\ b^1_3 \end{bmatrix}$$

$$W^2 = \begin{bmatrix} w^2_{11} & w^2_{12} & w^2_{13} \end{bmatrix} \quad b^2 = \begin{bmatrix} b^2_1 \end{bmatrix}$$

Output: $\hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2$

Neural network parameters:

$$W = \{W^1, b^1, W^2, b^2\}$$

Loss function (regression loss, same as before)**:**

Per-example: $L^i(W) = (\hat{y}^i - y^i)^2$

Over M examples: $L = \frac{1}{M} \sum_i L^i(W)$

Gradient of loss w.r.t. weights:

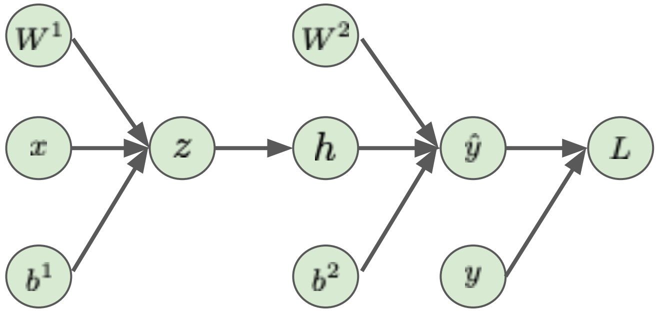
Function more complex -> now much harder to derive the expressions! Instead… computational graphs and backpropagation.

Computing gradients with backpropagation

Computing gradients with backpropagation

Network output: $\hat{y} = W^2(\sigma(W^1x + b^1)) + b^2$

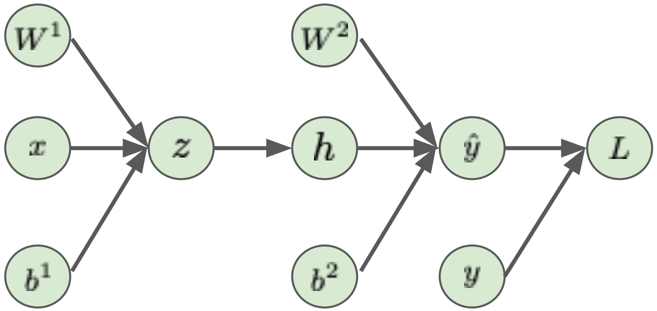
Think of computing loss function as staged computation of intermediate variables:



Computing gradients with backpropagation

Network output: $\hat{y} = W^2(\sigma(W^1x + b^1)) + b^2$

Think of computing loss function as staged computation of intermediate variables:



"Forward pass": $z = W^1x + b^1$

$$h = \sigma(z)$$

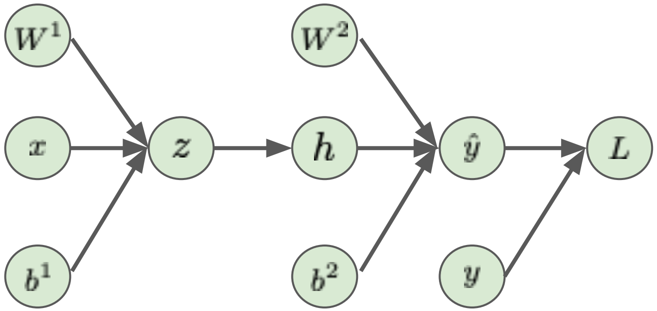
$$\hat{y} = W^2h + b^2$$

$$L = (\hat{y} - y)^2$$

Computing gradients with backpropagation

Network output: $\hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2$

Think of computing loss function as staged computation of intermediate variables:



"Forward pass": $z = W^1 x + b^1$

$$h = \sigma(z)$$

$$\hat{y} = W^2 h + b^2$$

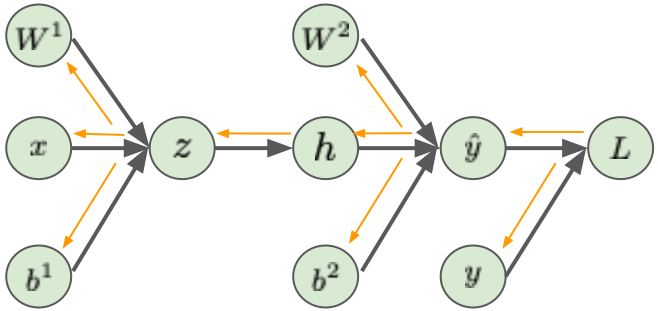
$$L = (\hat{y} - y)^2$$

Now, can use a repeated application of the chain rule, going backwards through the computational graph, to obtain the gradient of the loss with respect to each node of the computation graph.

Computing gradients with backpropagation

Network output: $\hat{y} = W^2(\sigma(W^1x + b^1)) + b^2$

Think of computing loss function as staged computation of intermediate variables:



"Forward pass":

$$z = W^1x + b^1$$
$$h = \sigma(z)$$
$$\hat{y} = W^2h + b^2$$
$$L = (\hat{y} - y)^2$$

Now, can use a repeated application of the chain rule, going backwards through the computational graph, to obtain the gradient of the loss with respect to each node of the computation graph.

"Backward pass":

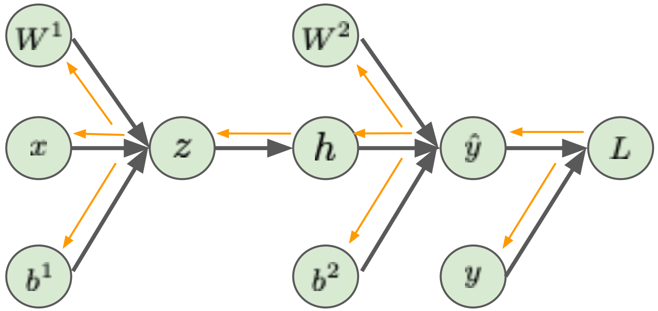
$$\frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$$
$$\frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W^2}$$
$$\frac{\partial L}{\partial H} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial H}$$
$$\frac{\partial L}{\partial Z} = \frac{\partial L}{\partial H} \frac{\partial H}{\partial Z}$$
$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial Z} \frac{\partial z}{\partial W^1}$$

(not all gradients shown)

Computing gradients with backpropagation

Network output: $\hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2$

Think of computing loss function as staged computation of intermediate variables:



"Forward pass":
$$z = W^1 x + b^1$$
$$h = \sigma(z)$$
$$\hat{y} = W^2 h + b^2$$
$$L = (\hat{y} - y)^2$$

Now, can use a repeated application of the chain rule, going backwards through the computational graph, to obtain the gradient of the loss with respect to each node of the computation graph.

"Backward pass": $\frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$ (not all gradients shown)

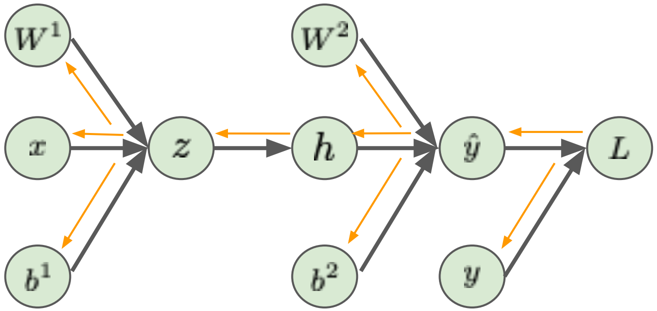
$$\frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W^2}$$
$$\frac{\partial L}{\partial H} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial H}$$
$$\frac{\partial L}{\partial Z} = \frac{\partial L}{\partial H} \frac{\partial H}{\partial Z}$$
$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial Z} \frac{\partial z}{\partial W^1}$$

Plug in from earlier computations via chain rule

Computing gradients with backpropagation

Network output: $\hat{y} = W^2(\sigma(W^1 x + b^1)) + b^2$

Think of computing loss function as staged computation of intermediate variables:



"Forward pass":
$$z = W^1 x + b^1$$
$$h = \sigma(z)$$
$$\hat{y} = W^2 h + b^2$$
$$L = (\hat{y} - y)^2$$

Now, can use a repeated application of the chain rule, going backwards through the computational graph, to obtain the gradient of the loss with respect to each node of the computation graph.

"Backward pass":
$$\frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$$
(not all gradients shown)

$$\frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial W^2}$$
$$\frac{\partial L}{\partial H} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial H}$$
$$\frac{\partial L}{\partial Z} = \frac{\partial L}{\partial H} \frac{\partial H}{\partial Z}$$
$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial Z} \frac{\partial z}{\partial W^1}$$

Plug in from earlier computations via chain rule

Local gradients to derive

Computing gradients with backpropagation

Key idea: Don't mathematically derive entire math expression for e.g. dL / dW^1. By writing it as nested applications of the chain rule, only have to derive simple "local" gradients representing relationships between connected nodes of the graph (e.g. dH / dW^1).

Computing gradients with backpropagation

Key idea: Don't mathematically derive entire math expression for e.g. dL / dW^1. By writing it as nested applications of the chain rule, only have to derive simple "local" gradients representing relationships between connectected nodes of the graph (e.g. dH / dW^1).

Can use more or less intermediate variables to control how difficult local gradients are to derive!

Training our two-layer neural network in code, using backpropagation

```
# initialize model parameters to be learned
W1 = np.random.rand(input_dim, hid_dim)
W2 = np.random.rand(hid_dim, output_dim)
b1 = np.random.rand(1, hid_dim)
b2 = np.random.rand(1, output_dim)

# perform gradient descent
step_size = 1e-2

while(keep_training)

    # forward pass, computing loss
    Z_curr = X.dot(W1) + b1
    H_curr = sigmoid_array(Z_curr)
    Y_curr = H_curr.dot(W2) + b2
    loss = np.sum(np.square(Y_curr - Y)) / num_examples

    # backward pass, computing gradients of loss with respect to each
    # variable in the computation graph
    d_Y_curr = 2*(Y_curr - Y) / num_examples
    d_H_curr = d_Y_curr.dot(W2.T)
    d_W2 = H_curr.T.dot(d_Y_curr)
    d_b2 = d_Y_curr
    d_Z_curr = d_H_curr * sigmoid_array(Z_curr)*(1-sigmoid_array(Z_curr))
    d_X = d_Z_curr.dot(W1.T)
    d_W1 = d_X.T.dot(d_Z_curr)
    d_b1 = d_Y_curr

    # perform gradient update
    W1 = W1 - step_size * d_W1
    b1 = b1 - step_size * d_b1
    W2 = W2 - step_size * d_W2
    b2 = b2 - step_size * d_b2
```

Training our two-layer neural network in code, using backpropagation

```
# initialize model parameters to be learned
W1 = np.random.rand(input_dim, hid_dim)
W2 = np.random.rand(hid_dim, output_dim)
b1 = np.random.rand(1, hid_dim)
b2 = np.random.rand(1, output_dim)

# perform gradient descent
step_size = 1e-2

while(keep_training)

    # forward pass, computing loss
    Z_curr = X.dot(W1) + b1
    H_curr = sigmoid_array(Z_curr)
    Y_curr = H_curr.dot(W2) + b2
    loss = np.sum(np.square(Y_curr - Y)) / num_examples

    # backward pass, computing gradients of loss with respect to each
    # variable in the computation graph
    d_Y_curr = 2*(Y_curr - Y) / num_examples
    d_H_curr = d_Y_curr.dot(W2.T)
    d_W2 = H_curr.T.dot(d_Y_curr)
    d_b2 = d_Y_curr
    d_Z_curr = d_H_curr * sigmoid_array(Z_curr)*(1-sigmoid_array(Z_curr))
    d_X = d_Z_curr.dot(W1.T)
    d_W1 = d_X.T.dot(d_Z_curr)
    d_b1 = d_Y_curr

    # perform gradient update
    W1 = W1 - step_size * d_W1
    b1 = b1 - step_size * d_b1
    W2 = W2 - step_size * d_W2
    b2 = b2 - step_size * d_b2
```

Initialize model parameters

Training our two-layer neural network in code, using backpropagation

```
# initialize model parameters to be learned
W1 = np.random.rand(input_dim, hid_dim)
W2 = np.random.rand(hid_dim, output_dim)
b1 = np.random.rand(1, hid_dim)
b2 = np.random.rand(1, output_dim)

# perform gradient descent
step_size = 1e-2

while(keep_training)

    # forward pass, computing loss
    Z_curr = X.dot(W1) + b1
    H_curr = sigmoid_array(Z_curr)
    Y_curr = H_curr.dot(W2) + b2
    loss = np.sum(np.square(Y_curr - Y)) / num_examples

    # backward pass, computing gradients of loss with respect to each
    # variable in the computation graph
    d_Y_curr = 2*(Y_curr - Y) / num_examples
    d_H_curr = d_Y_curr.dot(W2.T)
    d_W2 = H_curr.T.dot(d_Y_curr)
    d_b2 = d_Y_curr
    d_Z_curr = d_H_curr * sigmoid_array(Z_curr)*(1-sigmoid_array(Z_curr))
    d_X = d_Z_curr.dot(W1.T)
    d_W1 = d_X.T.dot(d_Z_curr)
    d_b1 = d_Y_curr

    # perform gradient update
    W1 = W1 - step_size * d_W1
    b1 = b1 - step_size * d_b1
    W2 = W2 - step_size * d_W2
    b2 = b2 - step_size * d_b2
```

Forward pass

Training our two-layer neural network in code, using backpropagation

```
# initialize model parameters to be learned
W1 = np.random.rand(input_dim, hid_dim)
W2 = np.random.rand(hid_dim, output_dim)
b1 = np.random.rand(1, hid_dim)
b2 = np.random.rand(1, output_dim)

# perform gradient descent
step_size = 1e-2

while(keep_training)

    # forward pass, computing loss
    Z_curr = X.dot(W1) + b1
    H_curr = sigmoid_array(Z_curr)
    Y_curr = H_curr.dot(W2) + b2
    loss = np.sum(np.square(Y_curr - Y)) / num_examples

    # backward pass, computing gradients of loss with respect to each
    # variable in the computation graph
    d_Y_curr = 2*(Y_curr - Y) / num_examples
    d_H_curr = d_Y_curr.dot(W2.T)
    d_W2 = H_curr.T.dot(d_Y_curr)
    d_b2 = d_Y_curr
    d_Z_curr = d_H_curr * sigmoid_array(Z_curr)*(1-sigmoid_array(Z_curr))
    d_X = d_Z_curr.dot(W1.T)
    d_W1 = d_X.T.dot(d_Z_curr)
    d_b1 = d_Y_curr

    # perform gradient update
    W1 = W1 - step_size * d_W1
    b1 = b1 - step_size * d_b1
    W2 = W2 - step_size * d_W2
    b2 = b2 - step_size * d_b2
```

Backward pass

Training our two-layer neural network in code, using backpropagation

```
# initialize model parameters to be learned
W1 = np.random.rand(input_dim, hid_dim)
W2 = np.random.rand(hid_dim, output_dim)
b1 = np.random.rand(1, hid_dim)
b2 = np.random.rand(1, output_dim)

# perform gradient descent
step_size = 1e-2

while(keep_training)

    # forward pass, computing loss
    Z_curr = X.dot(W1) + b1
    H_curr = sigmoid_array(Z_curr)
    Y_curr = H_curr.dot(W2) + b2
    loss = np.sum(np.square(Y_curr - Y)) / num_examples

    # backward pass, computing gradients of loss with respect to each
    # variable in the computation graph
    d_Y_curr = 2*(Y_curr - Y) / num_examples
    d_H_curr = d_Y_curr.dot(W2.T)
    d_W2 = H_curr.T.dot(d_Y_curr)
    d_b2 = d_Y_curr
    d_Z_curr = d_H_curr * sigmoid_array(Z_curr)*(1-sigmoid_array(Z_curr))
    d_X = d_Z_curr.dot(W1.T)
    d_W1 = d_X.T.dot(d_Z_curr)
    d_b1 = d_Y_curr

    # perform gradient update
    W1 = W1 - step_size * d_W1
    b1 = b1 - step_size * d_b1
    W2 = W2 - step_size * d_W2
    b2 = b2 - step_size * d_b2
```

Upstream gradient

Downstream gradient

Backward pass

Training our two-layer neural network in code, using backpropagation

```
# initialize model parameters to be learned
W1 = np.random.rand(input_dim, hid_dim)
W2 = np.random.rand(hid_dim, output_dim)
b1 = np.random.rand(1, hid_dim)
b2 = np.random.rand(1, output_dim)

# perform gradient descent
step_size = 1e-2

while(keep_training)

    # forward pass, computing loss
    Z_curr = X.dot(W1) + b1
    H_curr = sigmoid_array(Z_curr)
    Y_curr = H_curr.dot(W2) + b2
    loss = np.sum(np.square(Y_curr - Y)) / num_examples

    # backward pass, computing gradients of loss with respect to each
    # variable in the computation graph
    d_Y_curr = 2*(Y_curr - Y) / num_examples
    d_H_curr = d_Y_curr.dot(W2.T)
    d_W2 = H_curr.T.dot(d_Y_curr)
    d_b2 = d_Y_curr
    d_Z_curr = d_H_curr * sigmoid_array(Z_curr)*(1-sigmoid_array(Z_curr))
    d_X = d_Z_curr.dot(W1.T)
    d_W1 = d_X.T.dot(d_Z_curr)
    d_b1 = d_Y_curr

    # perform gradient update
    W1 = W1 - step_size * d_W1
    b1 = b1 - step_size * d_b1
    W2 = W2 - step_size * d_W2
    b2 = b2 - step_size * d_b2
```

Gradient update

Deep learning software frameworks

- Makes our lives easier by providing implementations and higher-level abstractions of many components for deep learning, and running them on GPUs:
  - Dataset batching, model definition, gradient computation, optimization, etc.

Deep learning software frameworks

- Makes our lives easier by providing implementations and higher-level abstractions of many components for deep learning, and running them on GPUs:
    - Dataset batching, model definition, gradient computation, optimization, etc.

- Automatic differentiation: if we define nodes in a computational graph, will automatically implement backpropagation for us
    - Supports many common operations with local gradients already implemented
    - Can still define custom operations

Deep learning software frameworks

- Makes our lives easier by providing implementations and higher-level abstractions of many components for deep learning, and running them on GPUs:
    - Dataset batching, model definition, gradient computation, optimization, etc.

- Automatic differentiation: if we define nodes in a computational graph, will automatically implement backpropagation for us
    - Supports many common operations with local gradients already implemented
    - Can still define custom operations

- A number of popular options, e.g. Tensorflow and PyTorch. Recent stable versions (TF 2.0, PyTorch 1.3) work largely in a similar fashion (not necessarily true for earlier versions). We will use Tensorflow 2.0 in this class.

Deep learning software frameworks

- Makes our lives easier by providing implementations and higher-level abstractions of many components for deep learning, and running them on GPUs:
    - Dataset batching, model definition, gradient computation, optimization, etc.

- Automatic differentiation: if we define nodes in a computational graph, will automatically implement backpropagation for us
    - Supports many common operations with local gradients already implemented
    - Can still define custom operations

- A number of popular options, e.g. Tensorflow and PyTorch. Recent stable versions (TF 2.0, PyTorch 1.3) work largely in a similar fashion (not necessarily true for earlier versions). We will use Tensorflow 2.0 in this class.

- More next Friday, Sept 25, during our Tensorflow section.

Training our two-layer neural network in code, in Tensorflow 2.0

```
# Our (X,Y) training set converted to TF tensors
X_tf = tf.convert_to_tensor(X, np.float32)
Y_tf = tf.convert_to_tensor(Y, np.float32)
# Create a TF dataset with specified minibatch size
batch_size = 50
dataset = tf.data.Dataset.from_tensor_slices((X_tf, Y_tf))
dataset = dataset.batch(batch_size)

# initialize model parameters to be learned
W1 = tf.Variable(tf.random.uniform((input_dim, hid_dim)))
W2 = tf.Variable(tf.random.uniform((hid_dim, output_dim)))
b1 = tf.Variable(tf.random.uniform((1, hid_dim)))
b2 = tf.Variable(tf.random.uniform((1, output_dim)))

# perform gradient descent
epochs = 5000
optimizer = tf.optimizers.SGD(learning_rate=1e-2)
losses = []

for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:

            # forward pass
            Z_batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(Z_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

Training our two-layer neural network in code, in Tensorflow 2.0

```
# Our (X,Y) training set converted to TF tensors
X_tf = tf.convert_to_tensor(X, np.float32)
Y_tf = tf.convert_to_tensor(Y, np.float32)
# Create a TF dataset with specified minibatch size
batch_size = 50
dataset = tf.data.Dataset.from_tensor_slices((X_tf, Y_tf))
dataset = dataset.batch(batch_size)

# initialize model parameters to be learned
W1 = tf.Variable(tf.random.uniform((input_dim, hid_dim)))
W2 = tf.Variable(tf.random.uniform((hid_dim, output_dim)))
b1 = tf.Variable(tf.random.uniform((1, hid_dim)))
b2 = tf.Variable(tf.random.uniform((1, output_dim)))

# perform gradient descent
epochs = 5000
optimizer = tf.optimizers.SGD(learning_rate=1e-2)
losses = []

for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:

            # forward pass
            Z_batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(Z_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

Convert data to TF tensors, create a TF dataset

Training our two-layer neural network in code, in Tensorflow 2.0

```
# Our (X,Y) training set converted to TF tensors
X_tf = tf.convert_to_tensor(X, np.float32)
Y_tf = tf.convert_to_tensor(Y, np.float32)
# Create a TF dataset with specified minibatch size
batch_size = 50
dataset = tf.data.Dataset.from_tensor_slices((X_tf, Y_tf))
dataset = dataset.batch(batch_size)

# initialize model parameters to be learned
W1 = tf.Variable(tf.random.uniform((input_dim, hid_dim)))
W2 = tf.Variable(tf.random.uniform((hid_dim, output_dim)))
b1 = tf.Variable(tf.random.uniform((1, hid_dim)))
b2 = tf.Variable(tf.random.uniform((1, output_dim)))

# perform gradient descent
epochs = 5000
optimizer = tf.optimizers.SGD(learning_rate=1e-2)
losses = []

for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:

            # forward pass
            Z_batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(Z_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

Initialize parameters to be learned as tf.Variable -> allows them to receive gradient updates during optimization

Training our two-layer neural network in code, in Tensorflow 2.0

```
# Our (X,Y) training set converted to TF tensors
X_tf = tf.convert_to_tensor(X, np.float32)
Y_tf = tf.convert_to_tensor(Y, np.float32)
# Create a TF dataset with specified minibatch size
batch_size = 50
dataset = tf.data.Dataset.from_tensor_slices((X_tf, Y_tf))
dataset = dataset.batch(batch_size)

# initialize model parameters to be learned
W1 = tf.Variable(tf.random.uniform((input_dim, hid_dim)))
W2 = tf.Variable(tf.random.uniform((hid_dim, output_dim)))
b1 = tf.Variable(tf.random.uniform((1, hid_dim)))
b2 = tf.Variable(tf.random.uniform((1, output_dim)))

# perform gradient descent
epochs = 5000
optimizer = tf.optimizers.SGD(learning_rate=1e-2)
losses = []

for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:

            # forward pass
            Z_batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(Z_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

Initialize a TF optimizer

Training our two-layer neural network in code, in Tensorflow 2.0

```
# Our (X,Y) training set converted to TF tensors
X_tf = tf.convert_to_tensor(X, np.float32)
Y_tf = tf.convert_to_tensor(Y, np.float32)
# Create a TF dataset with specified minibatch size
batch_size = 50
dataset = tf.data.Dataset.from_tensor_slices((X_tf, Y_tf))
dataset = dataset.batch(batch_size)

# initialize model parameters to be learned
W1 = tf.Variable(tf.random.uniform((input_dim, hid_dim)))
W2 = tf.Variable(tf.random.uniform((hid_dim, output_dim)))
b1 = tf.Variable(tf.random.uniform((1, hid_dim)))
b2 = tf.Variable(tf.random.uniform((1, output_dim)))

# perform gradient descent
epochs = 5000
optimizer = tf.optimizers.SGD(learning_rate=1e-2)
losses = []

for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:

            # forward pass
            Z_batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(Z_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

All operations defined under the gradient tape will be used to construct a computational graph

Training our two-layer neural network in code, in Tensorflow 2.0

```
# Our (X,Y) training set converted to TF tensors
X_tf = tf.convert_to_tensor(X, np.float32)
Y_tf = tf.convert_to_tensor(Y, np.float32)
# Create a TF dataset with specified minibatch size
batch_size = 50
dataset = tf.data.Dataset.from_tensor_slices((X_tf, Y_tf))
dataset = dataset.batch(batch_size)

# initialize model parameters to be learned
W1 = tf.Variable(tf.random.uniform((input_dim, hid_dim)))
W2 = tf.Variable(tf.random.uniform((hid_dim, output_dim)))
b1 = tf.Variable(tf.random.uniform((1, hid_dim)))
b2 = tf.Variable(tf.random.uniform((1, output_dim)))

# perform gradient descent
epochs = 5000
optimizer = tf.optimizers.SGD(learning_rate=1e-2)
losses = []

for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:

            # forward pass
            Z_batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(Z_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

The computational graph for our two-layer neural network

Training our two-layer neural network in code, in Tensorflow 2.0

```
# Our (X,Y) training set converted to TF tensors
X_tf = tf.convert_to_tensor(X, np.float32)
Y_tf = tf.convert_to_tensor(Y, np.float32)
# Create a TF dataset with specified minibatch size
batch_size = 50
dataset = tf.data.Dataset.from_tensor_slices((X_tf, Y_tf))
dataset = dataset.batch(batch_size)

# initialize model parameters to be learned
W1 = tf.Variable(tf.random.uniform((input_dim, hid_dim)))
W2 = tf.Variable(tf.random.uniform((hid_dim, output_dim)))
b1 = tf.Variable(tf.random.uniform((1, hid_dim)))
b2 = tf.Variable(tf.random.uniform((1, output_dim)))

# perform gradient descent
epochs = 5000
optimizer = tf.optimizers.SGD(learning_rate=1e-2)
losses = []

for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:

            # forward pass
            Z_batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(Z_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

Evaluate gradients using automatic differentiation and perform gradient update

Also high level libraries built on top of Tensorflow, that provide even easier-to-use APIs:

In Tensorflow 2.0:

```
for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:

            # forward pass
            Z_batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(Z_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

In Keras:

```
keras_model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(units=3, activation='sigmoid', use_bias=True),
    tf.keras.layers.Dense(units=1, use_bias=True)
])
keras_model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=1e-2),
              loss='mse')
keras_model.fit(dataset, epochs=1000)
```

Also high level libraries built on top of Tensorflow, that provide even easier-to-use APIs:

Stack of layers

In Tensorflow 2.0:

```
for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:

            # forward pass
            Z_batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(Z_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

In Keras:

```
keras_model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(units=3, activation='sigmoid', use_bias=True),
    tf.keras.layers.Dense(units=1, use_bias=True)
])
keras_model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=1e-2),
          loss='mse')
keras_model.fit(dataset, epochs=1000)
```

Also high level libraries built on top of Tensorflow, that provide even easier-to-use APIs:

In Tensorflow 2.0:

```
for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:

            # forward pass
            Z_batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(Z_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

In Keras:

Fully-connected layer

```
keras_model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(units=3, activation='sigmoid', use_bias=True),
    tf.keras.layers.Dense(units=1, use_bias=True)
])
keras_model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=1e-2),
            loss='mse')
keras_model.fit(dataset, epochs=1000)
```

Also high level libraries built on top of Tensorflow, that provide even easier-to-use APIs:

Activation function and bias configurations included!

In Tensorflow 2.0:

```
for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:

            # forward pass
            Z_batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(Z_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

In Keras:

```
keras_model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(units=3, activation='sigmoid', use_bias=True),
    tf.keras.layers.Dense(units=1, use_bias=True)
])
keras_model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=1e-2),
                    loss='mse')
keras_model.fit(dataset, epochs=1000)
```

Also high level libraries built on top of Tensorflow, that provide even easier-to-use APIs:

In Tensorflow 2.0:

```
for epoch in range(epochs):
    for batch in dataset:
        X_batch, Y_batch = batch
        with tf.GradientTape() as tape:

            # forward pass
            Z_batch = tf.add(tf.matmul(X_batch, W1), b1)
            H_batch = tf.math.sigmoid(Z_batch)
            Out_batch = tf.add(tf.matmul(H_batch, W2), b2)
            loss = tf.losses.MSE(Y_batch, Out_batch)

            # backward pass and gradient update
            gradients = tape.gradient(loss, [W1, W2, b1, b2])
            optimizer.apply_gradients(zip(gradients, [W1, W2, b1, b2]))
```

In Keras:

```
keras_model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(units=3, activation='sigmoid', use_bias=True),
    tf.keras.layers.Dense(units=1, use_bias=True)
])
keras_model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=1e-2),
          loss='mse')
keras_model.fit(dataset, epochs=1000)
```

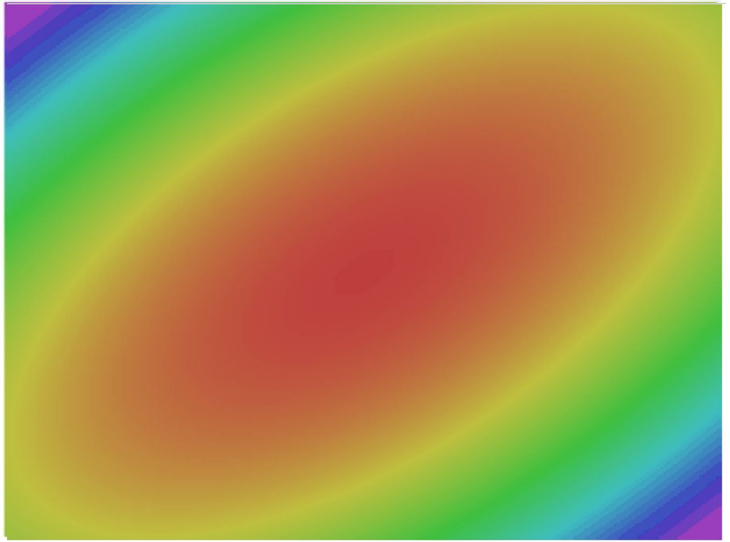
Specify hyperparameters for the training procedure

Design choices

Training hyperparameters: control knobs for the art of training neural networks

Optimization methods: SGD, SGD with momentum, RMSProp, Adam

- **Adam** is a good default choice in many cases; it often works ok even with constant learning rate
- **SGD+Momentum** can outperform Adam but may require more tuning of LR and schedule



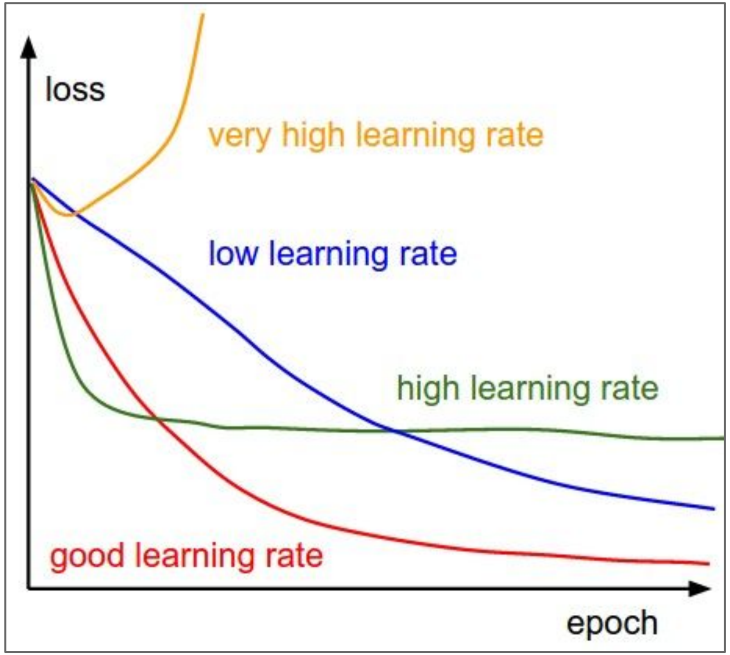
SGD

SGD+Momentum

RMSProp

Adam

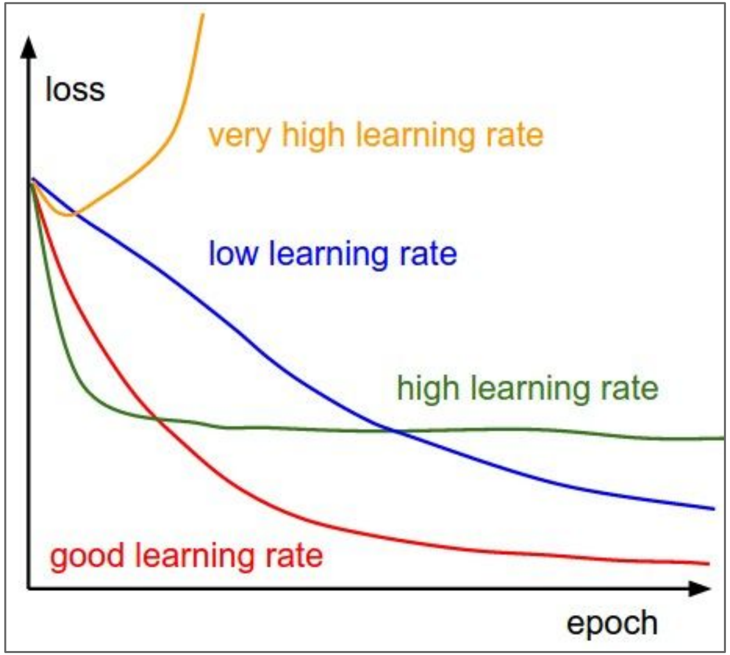
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



Q: Which one of these learning rates is best to use?

Slide credit: CS231n

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.

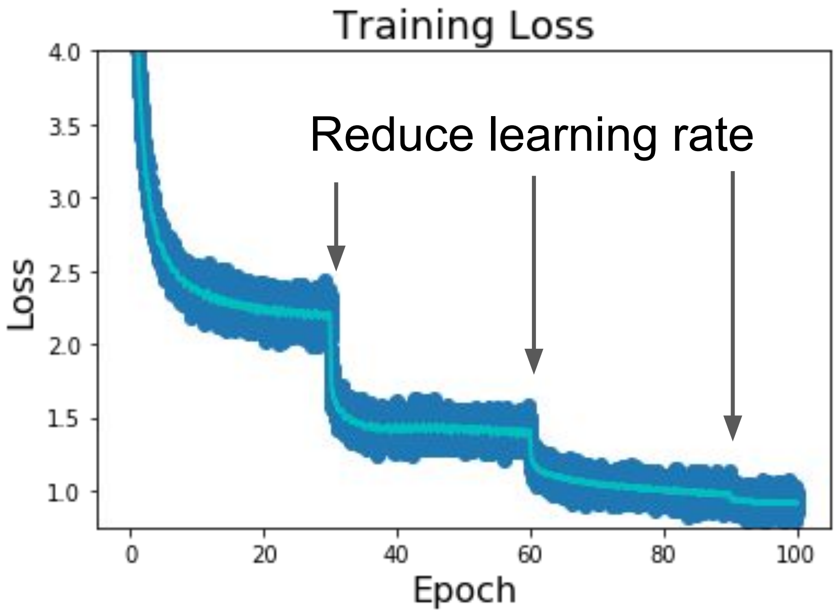


Q: Which one of these learning rates is best to use?

A: All of them! Start with large learning rate and decay over time

Slide credit: CS231n

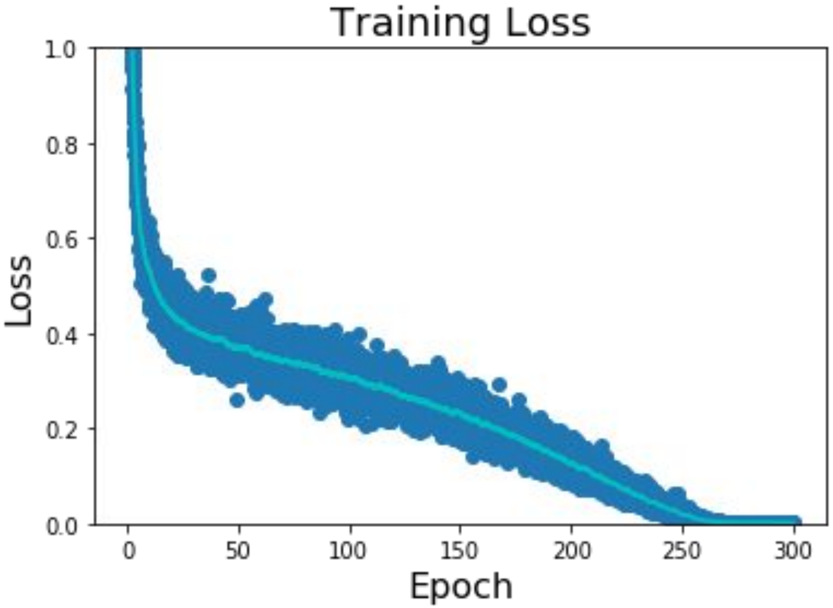
Learning rate decay



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Slide credit: CS231n

Learning rate decay



Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

Fancy decay schedules like cosine, linear, inverse sqrt.

Empirical rule of thumb: If you increase the batch size by N, also scale the initial learning rate by N

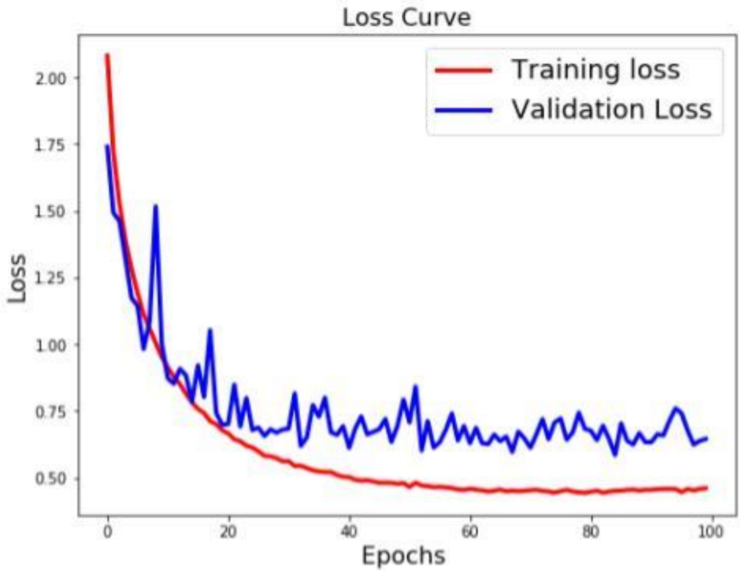
α_0 : Initial learning rate
α_t : Learning rate at epoch t
T : Total number of epochs

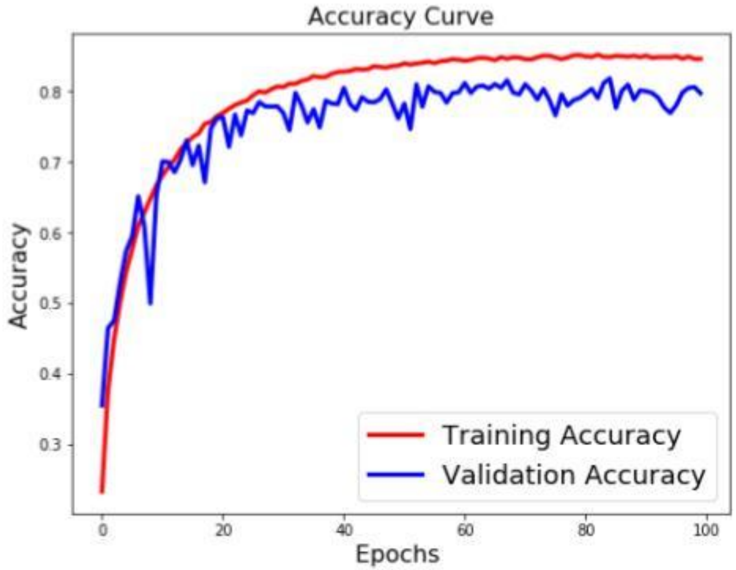
Loshchilov and Hutter, “SGDR: Stochastic Gradient Descent with Warm Restarts”, ICLR 2017
Radford et al, “Improving Language Understanding by Generative Pre-Training”, 2018
Feichtenhofer et al, “SlowFast Networks for Video Recognition”, arXiv 2018
Child at al, “Generating Long Sequences with Sparse Transformers”, arXiv 2019

Slide credit: CS231n

Monitor learning curves

Also useful to plot performance on final metric





Periodically evaluate validation loss

Figure credit: https://www.learnopencv.com/wp-content/uploads/2017/11/cnn-keras-curves-without-aug.jpg

Monitor learning curves

Training loss can be noisy. Using a scatter plot or plotting moving average can help better visualize trends.

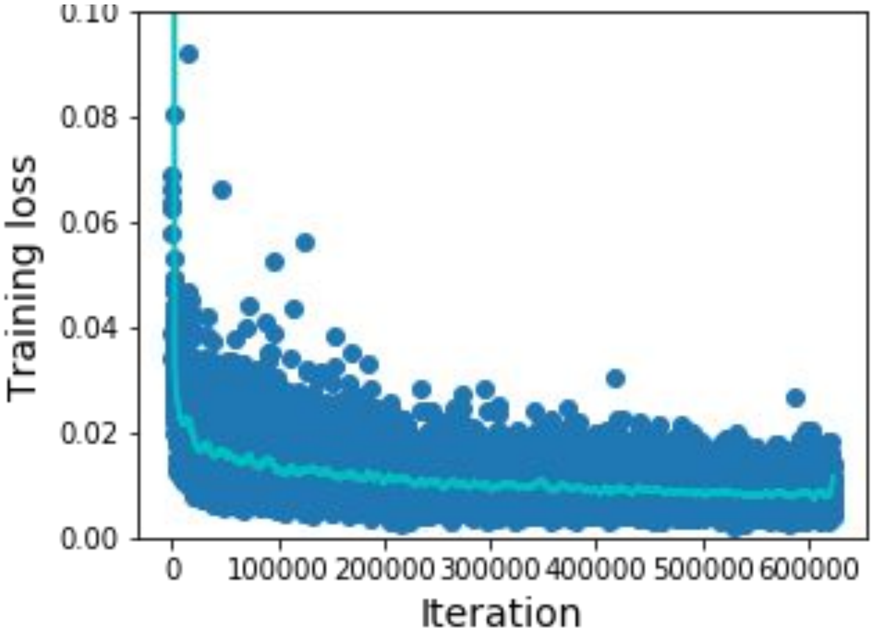
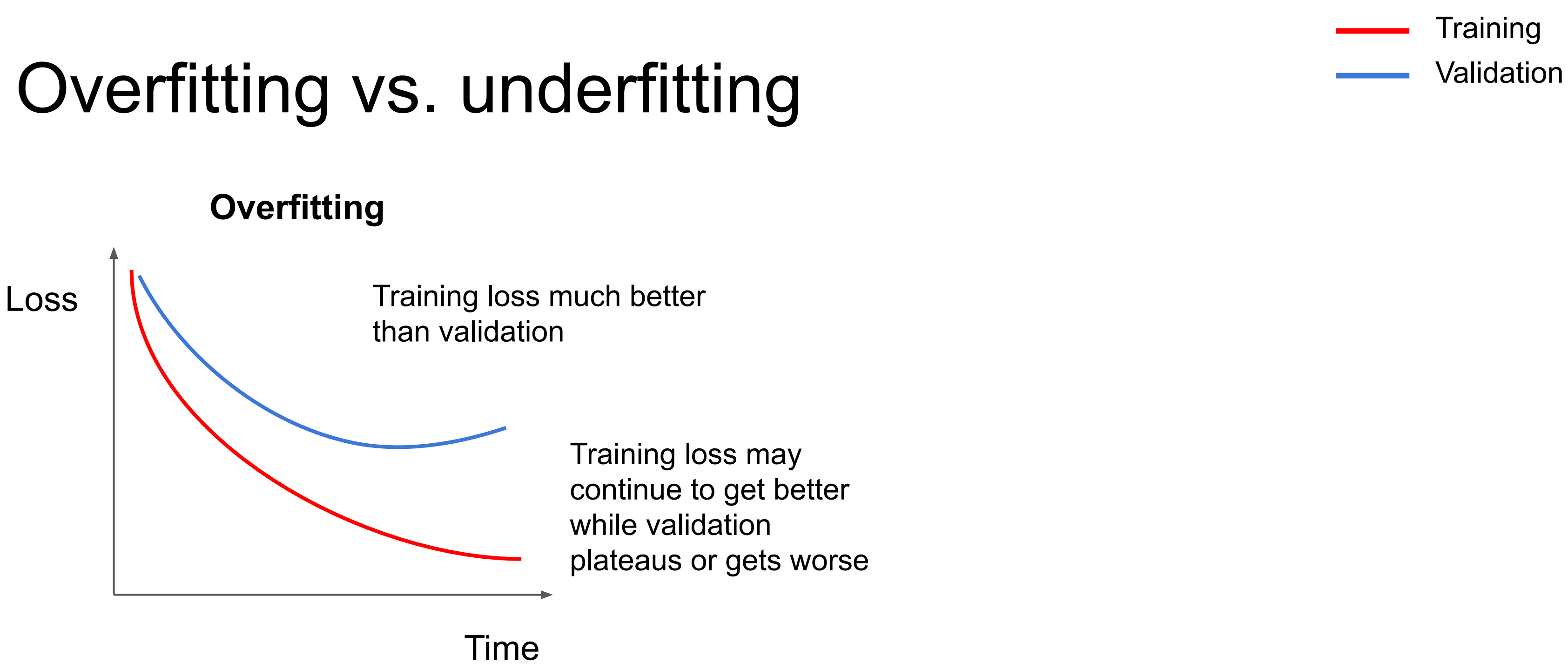


Figure credit: CS231n

Overfitting vs. underfitting



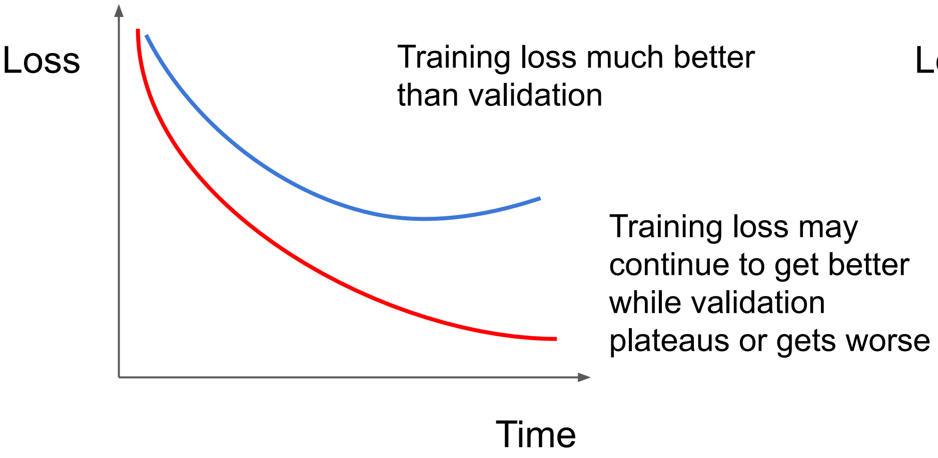
Training loss much better than validation

Training loss may continue to get better while validation plateaus or gets worse

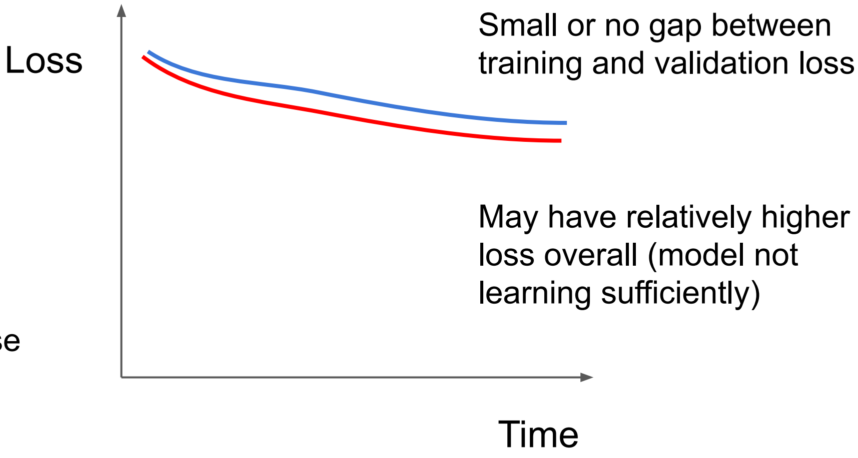
Overfitting vs. underfitting

Training
Validation

Overfitting



Underfitting



Overfitting vs. underfitting

Breakout session:

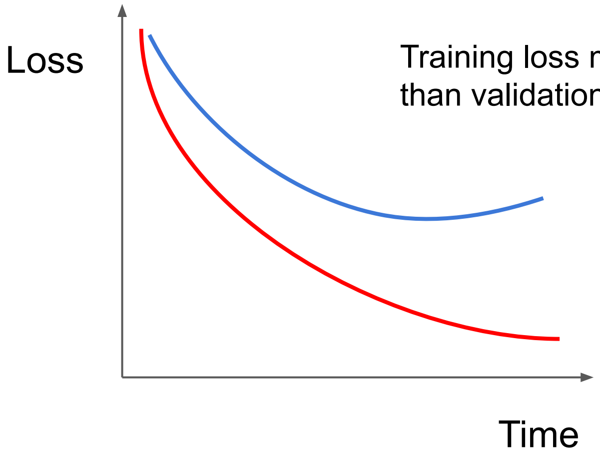
1x 5-minute breakout (~4 students each)

- Introduce yourself!
- What are some ways to combat overfitting?
- What are some ways to combat underfitting?

Training
Validation

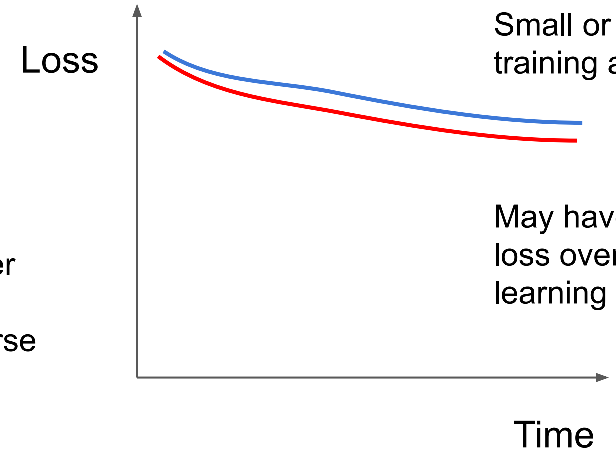
Overfitting vs. underfitting

Overfitting



Training loss may continue to get better while validation plateaus or gets worse

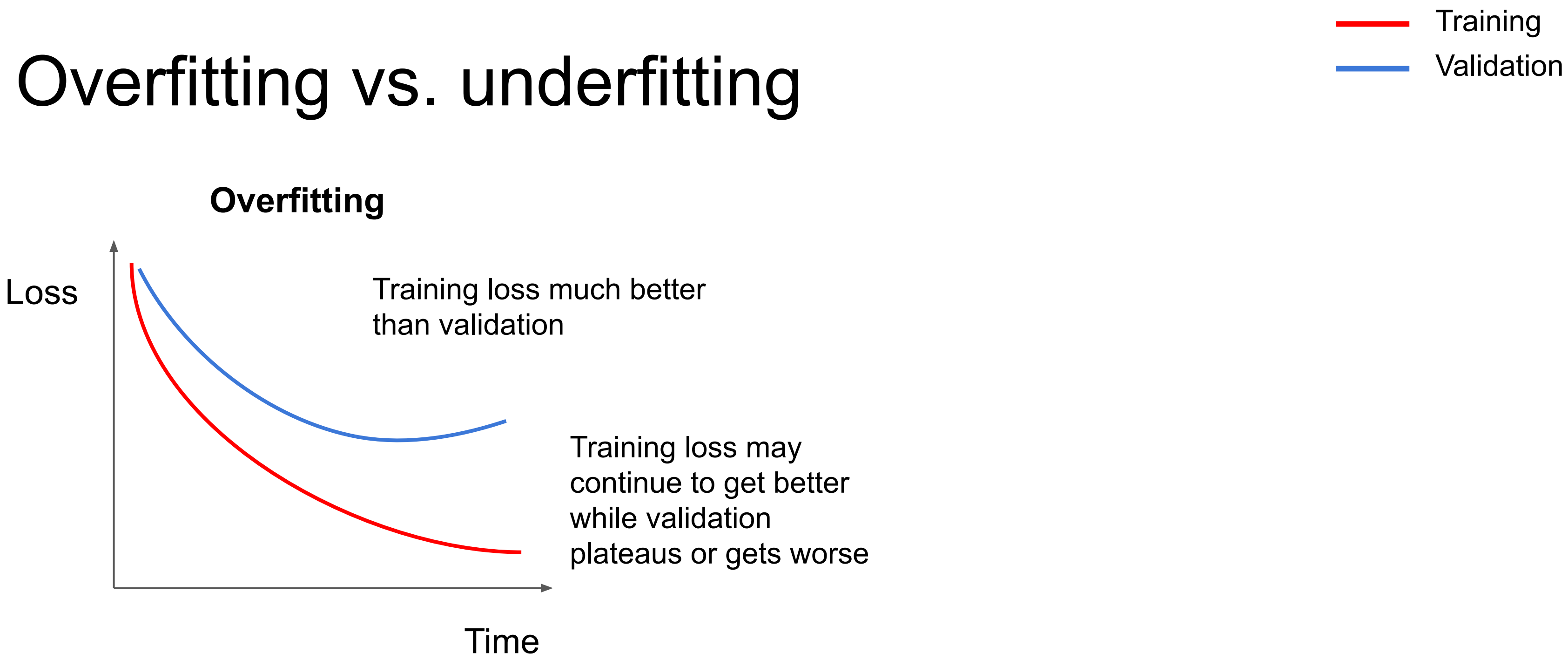
Underfitting



Small or no gap between training and validation loss

May have relatively higher loss overall (model not learning sufficiently)

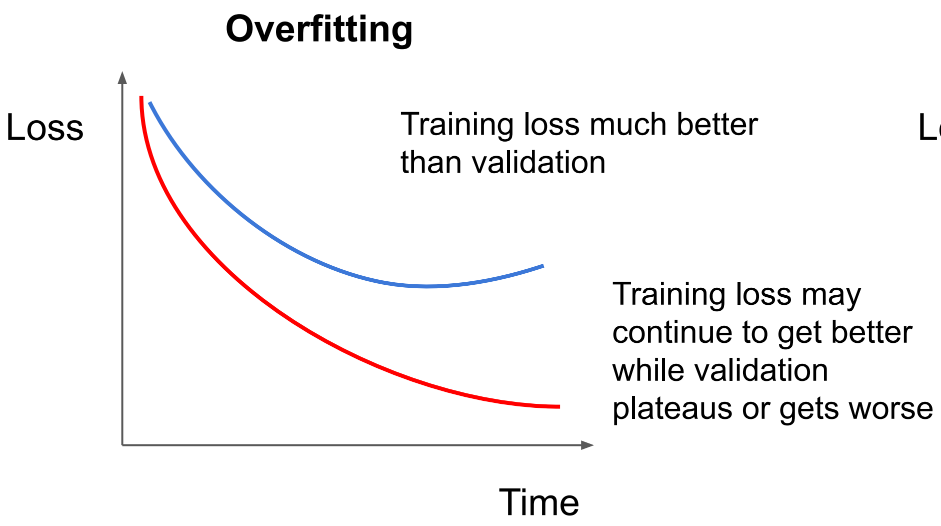
Overfitting vs. underfitting

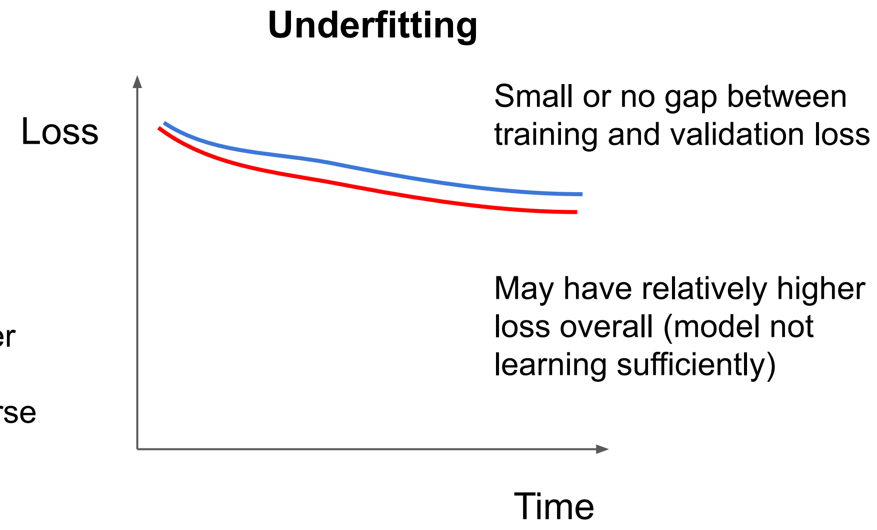


Model is “overfitting” to the training data. Best strategy: Increase data or regularize model. Second strategy: decrease model capacity (make simpler)

Training
Validation

Overfitting vs. underfitting





Model is “overfitting” to the training data. Best strategy: Increase data or regularize model. Second strategy: decrease model capacity (make simpler)

Model is not able to sufficiently learn to fit the data well. Best strategy: Increase complexity (e.g. size) of the model. Second strategy: make problem simpler (easier task, cleaner data)

Overfitting vs. underfitting: more intuition

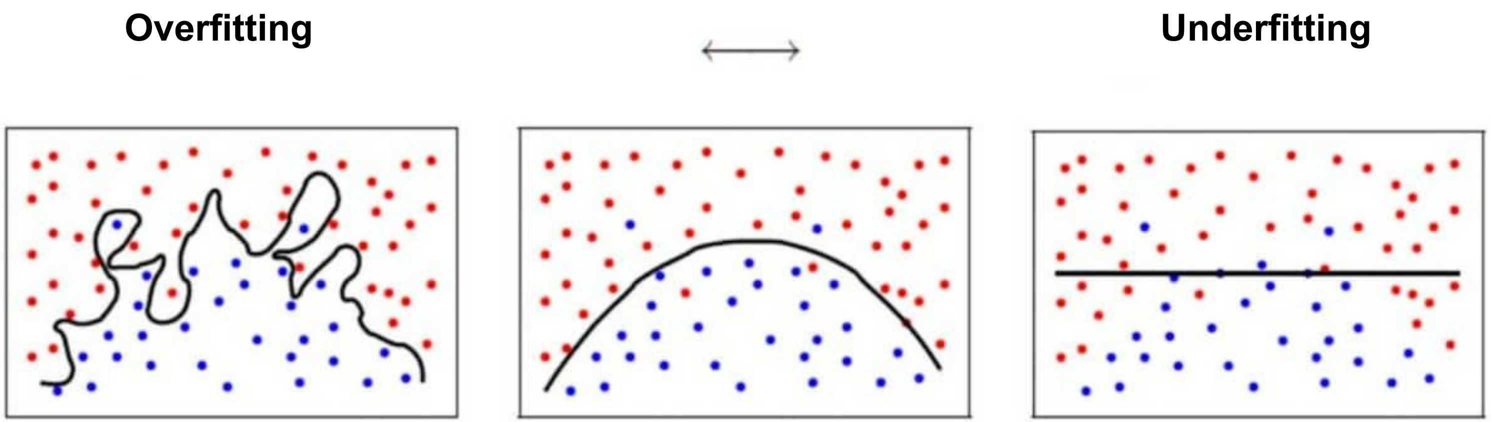
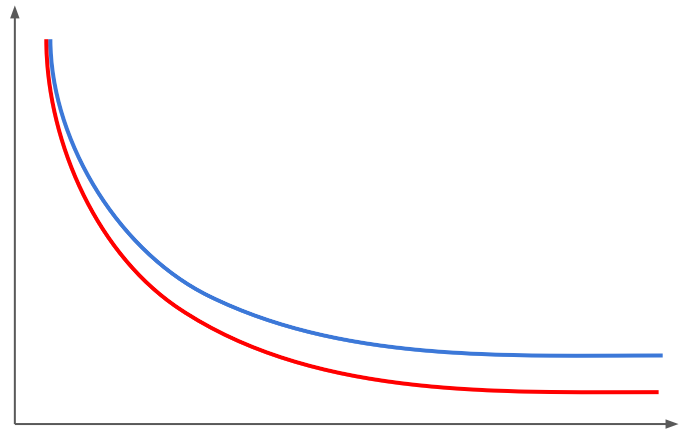


Figure credit: https://qph.fs.quoracdn.net/main-qimg-412c8556aacf7e25b86bba63e9e67ac6-c

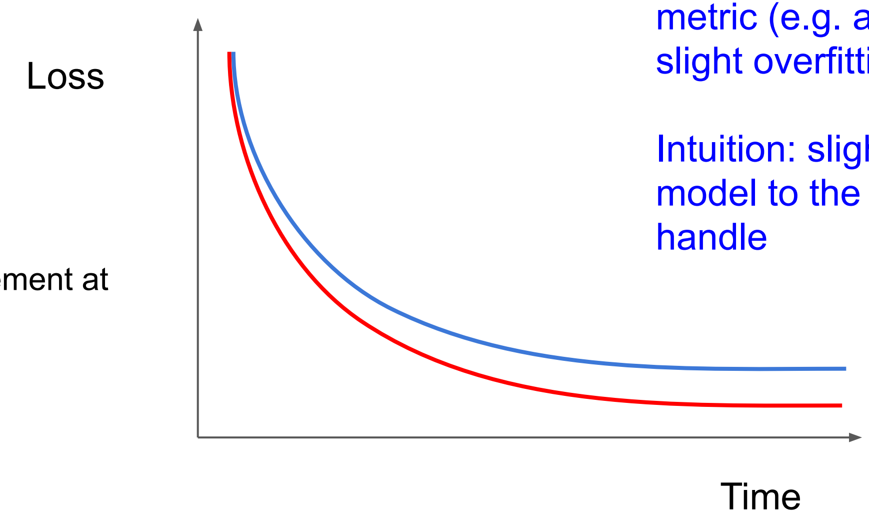
Healthy learning curves



Steep improvement at beginning

Continue to gradually improve

Healthy learning curves



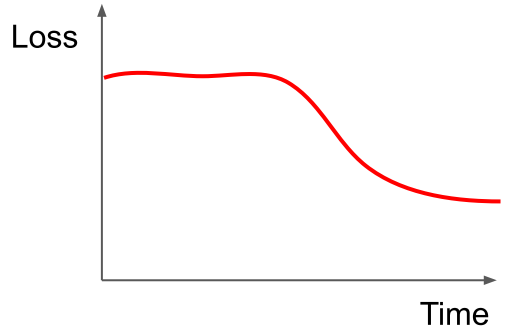
Steep improvement at beginning

Continue to gradually improve

In practice, models with best final metric (e.g. accuracy) often have slight overfitting.

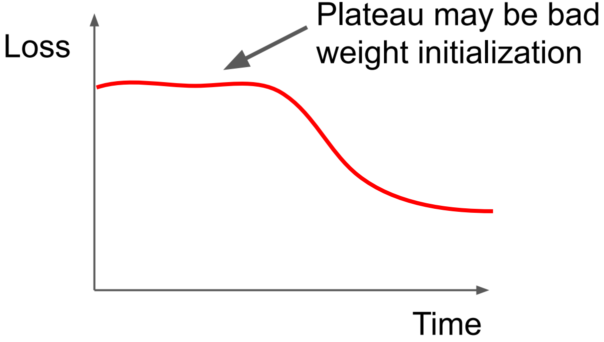
Intuition: slightly push complexity of model to the highest that the data can handle

More debugging

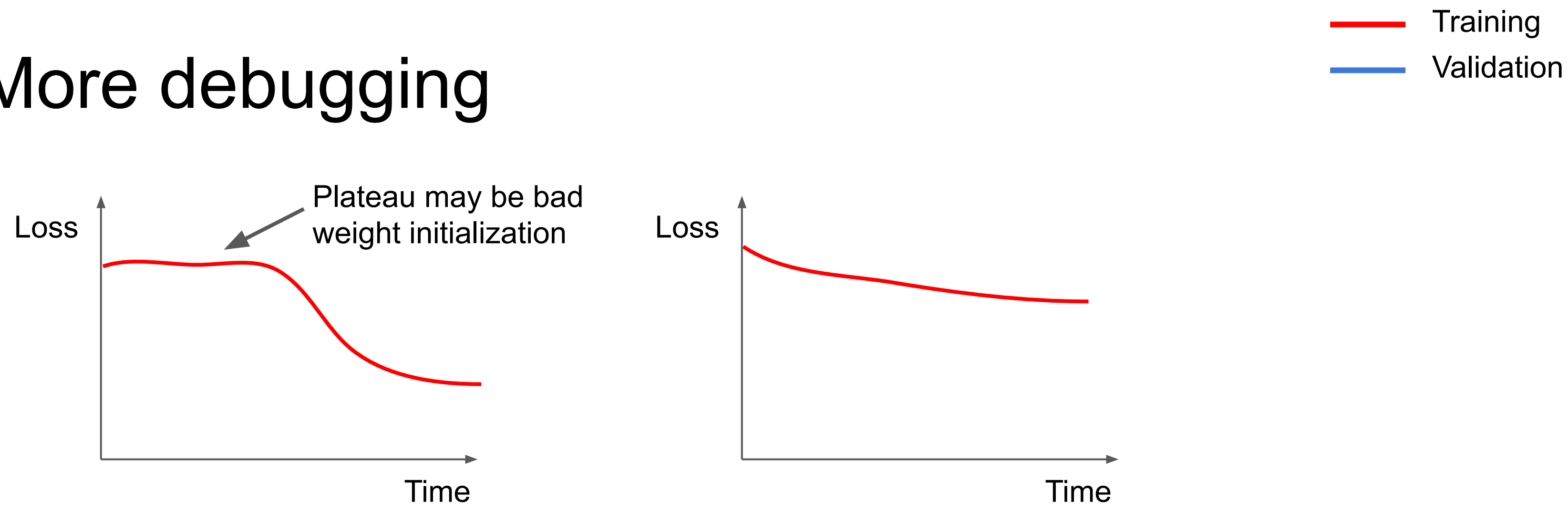


More debugging

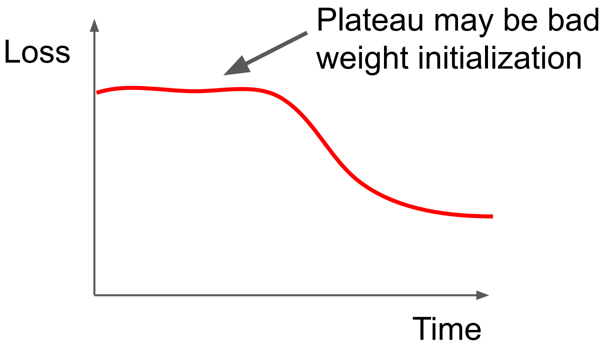
Training
Validation

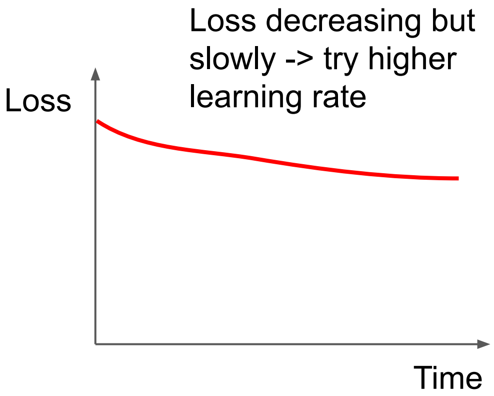


More debugging

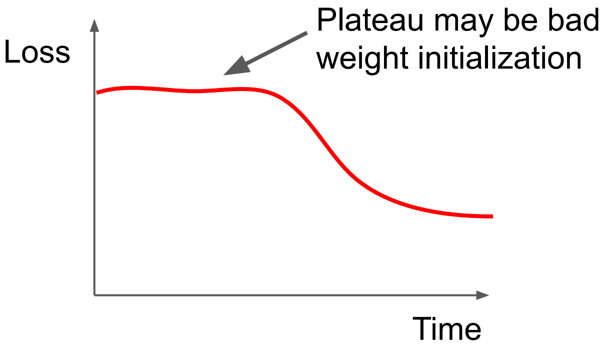


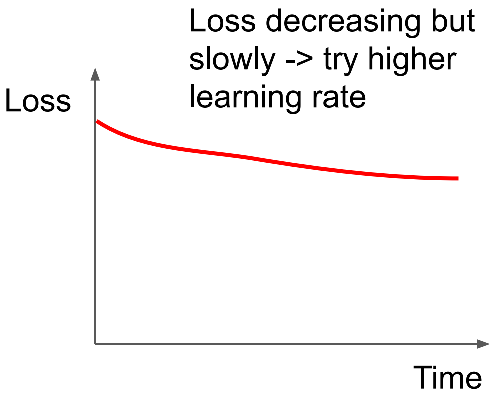
More debugging

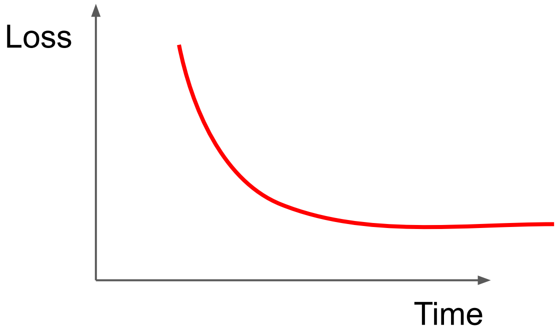




More debugging

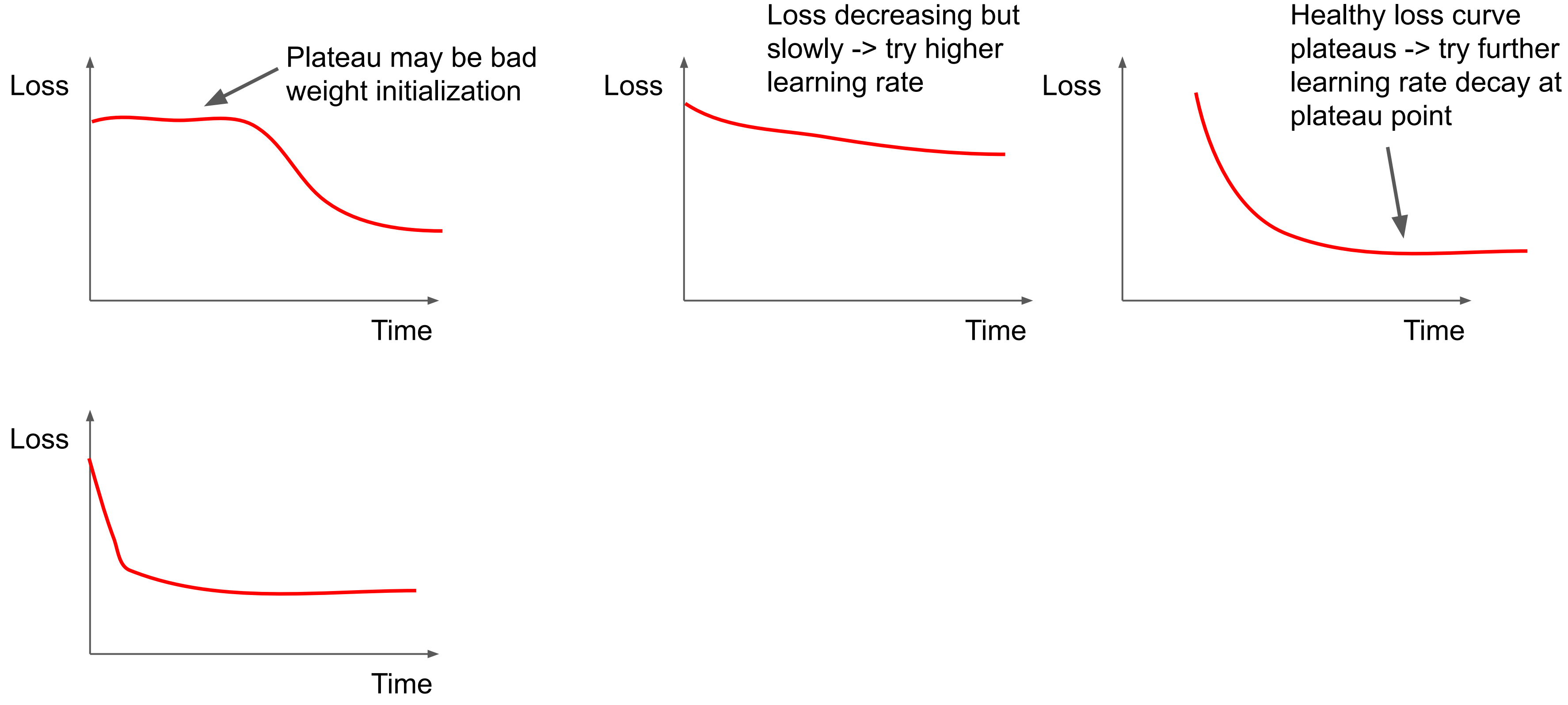






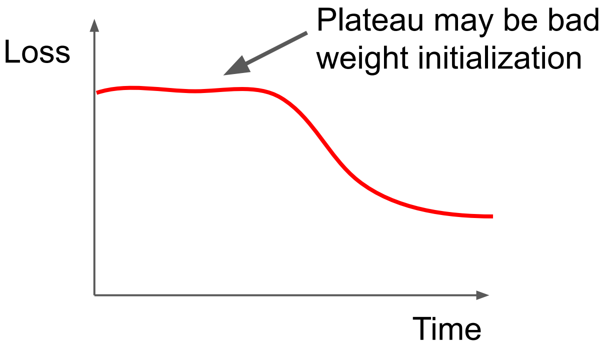
More debugging

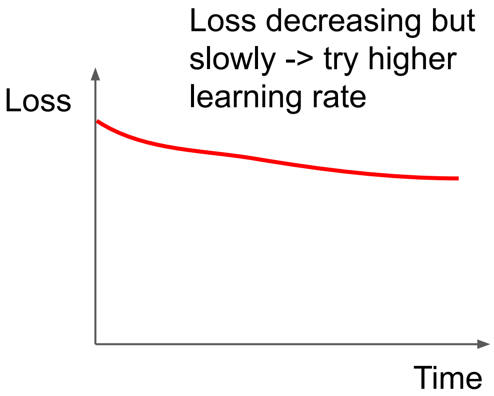
More debugging

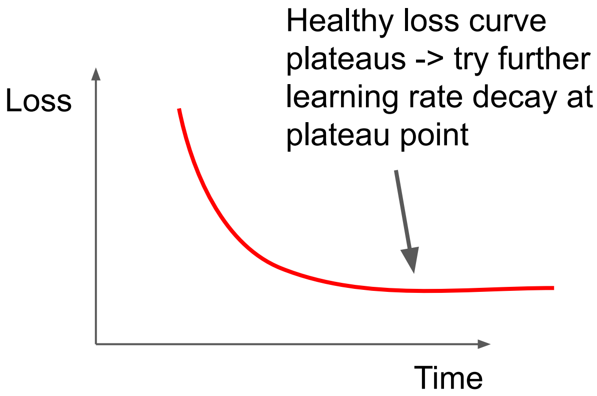


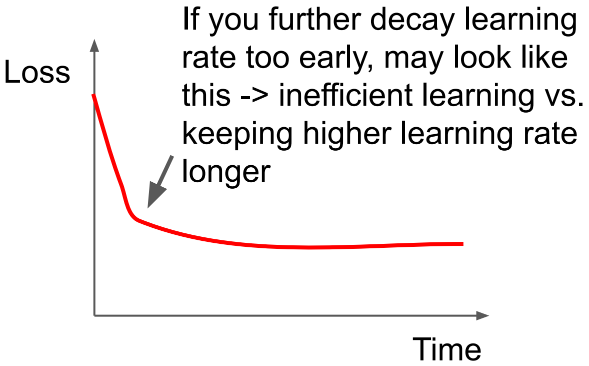
More debugging

Training
Validation

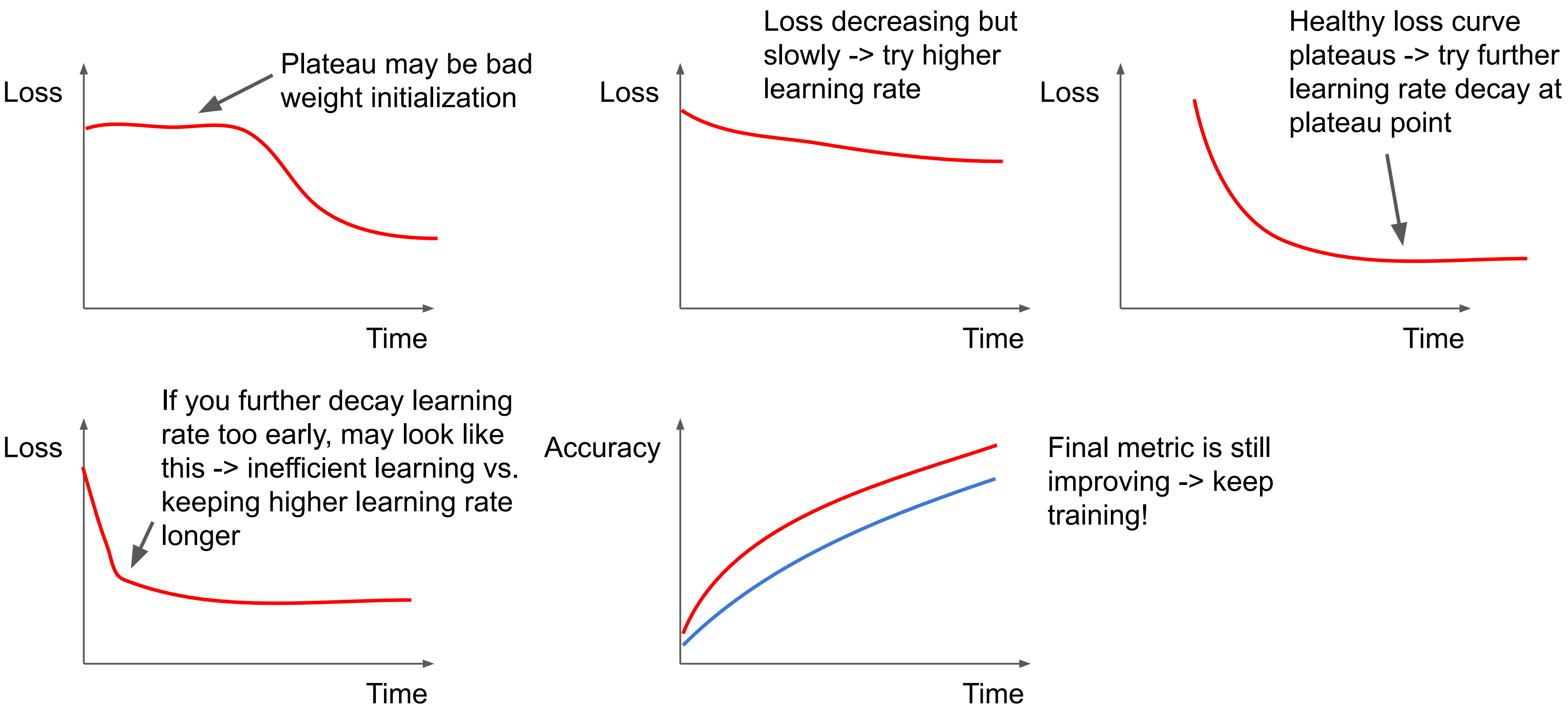




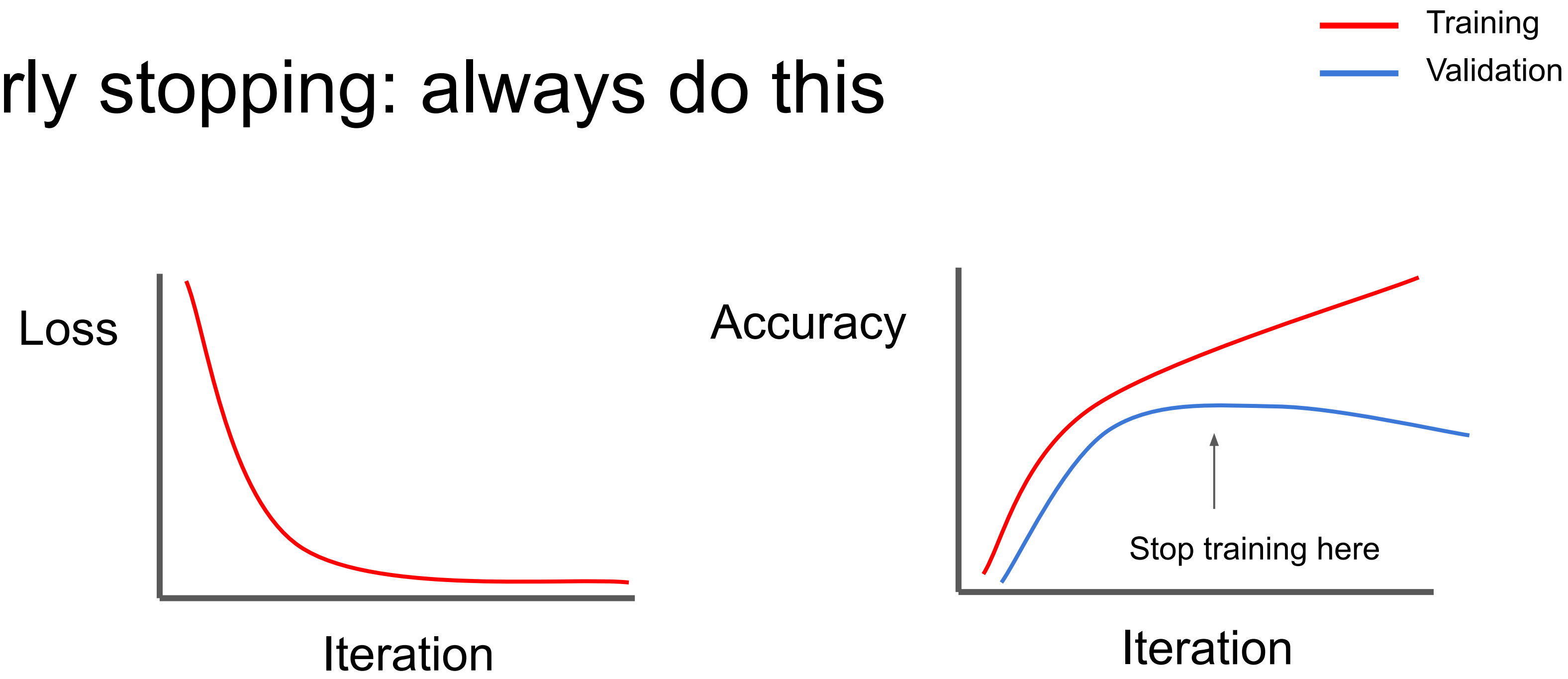




More debugging



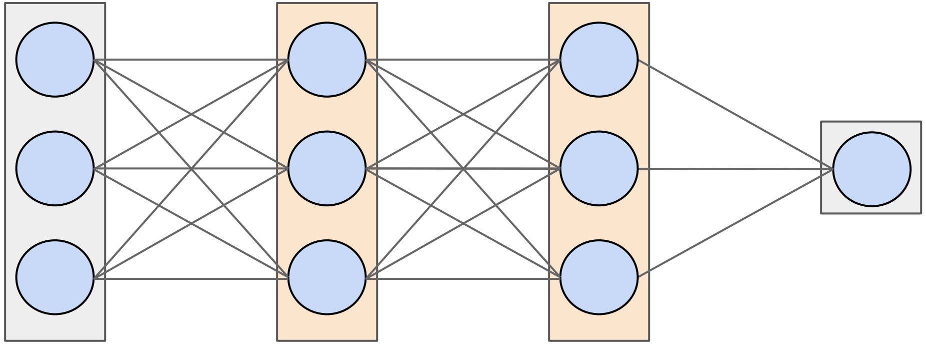
Early stopping: always do this



Stop training the model when accuracy on the validation set decreases
Or train for a long time, but always keep track of the model snapshot that worked best on val.

Slide credit: CS231n

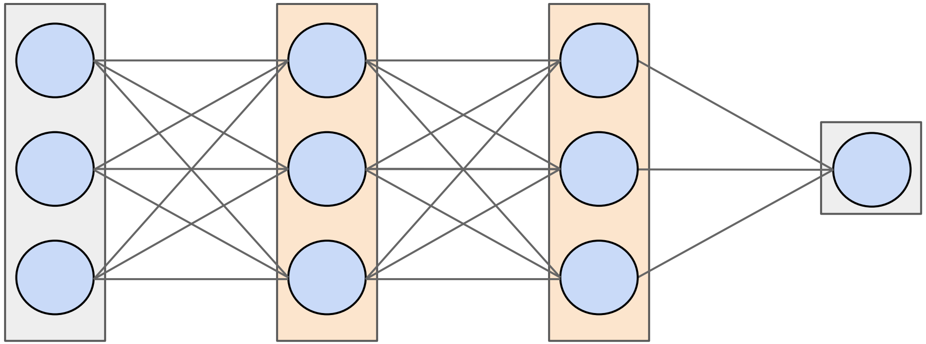
Design choices: network architectures



Major design choices:

- Architecture type (ResNet, DenseNet, etc. for CNNs)
- Depth (# layers)
- For MLPs, # neurons in each layer (hidden layer size)
- For CNNs, # filters, filter size, filter stride in each layer
- Look at argument options in Tensorflow when defining network layers

Design choices: network architectures



If trying to make network bigger (when underfitting) or smaller (when overfitting), network depth and hidden layer size best to adjust first. Don't waste too much time early on fiddling with choices that only minorly change architecture.

Major design choices:
- Architecture type (ResNet, DenseNet, etc. for CNNs)
- Depth (# layers)
- For MLPs, # neurons in each layer (hidden layer size)
- For CNNs, # filters, filter size, filter stride in each layer
- Look at argument options in Tensorflow when defining network layers

Design choices: regularization (loss term)

Remember optimizing loss functions, which express how well model fit training data, e.g.:

$$L_{regression} = \frac{1}{M}\sum_{i}(\hat{y}^i - y^i)^2$$

Design choices: regularization (loss term)

Remember optimizing loss functions, which express how well model fit training data, e.g.:

$$L_{regression} = \frac{1}{M}\sum_i (\hat{y}^i - y^i)^2$$

Regularization adds a term to this, to express preferences on the weights (that prevent it from fitting too well to the training data). Used to combat overfitting:

importance of reg. term

$$L = \underbrace{\frac{1}{M}\sum_i (\hat{y}^i - y^i)^2}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization loss}}$$

Data loss

Regularization loss

Design choices: regularization (loss term)

Remember optimizing loss functions, which express how well model fit training data, e.g.:

$$L_{regression} = \frac{1}{M}\sum_i (\hat{y}^i - y^i)^2$$

Regularization adds a term to this, to express preferences on the weights (that prevent it from fitting too well to the training data). Used to combat overfitting:

importance of reg. term

$$L = \underbrace{\frac{1}{M}\sum_i (\hat{y}^i - y^i)^2}_{\text{Data loss}} + \lambda \underbrace{R(W)}_{\text{Regularization loss}}$$

Examples

L2 regularization: $R(W) = \sum_k \sum_l W_{k,l}^2$ (weight decay)

L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2): $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

https://www.tensorflow.org/api_docs/python/tf/keras/regularizers

Design choices: regularization (loss term)

Remember optimizing loss functions, which express how well model fit training data, e.g.:

$$L_{regression} = \frac{1}{M}\sum_i (\hat{y}^i - y^i)^2$$

Regularization adds a term to this, to express preferences on the weights (that prevent it from fitting too well to the training data). Used to combat overfitting:

importance of reg. term

$$L = \underbrace{\frac{1}{M}\sum_i (\hat{y}^i - y^i)^2}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization loss}}$$

L2 most popular: low loss when all weights are relatively small. More strongly penalizes large weights vs L1. Expresses preference for simple models (need large coefficients to fit a function to extreme outlier values).

Examples

L2 regularization: $R(W) = \sum_k \sum_l W_{k,l}^2$ (weight decay)

L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2): $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

https://www.tensorflow.org/api_docs/python/tf/keras/regularizers

Design choices: regularization (loss term)

Remember optimizing loss functions, which express how well model fit training data, e.g.:

$$L_{regression} = \frac{1}{M}\sum_i (\hat{y}^i - y^i)^2$$

Regularization adds a term to this, to express preferences on the weights (that prevent it from fitting too well to the training data). Used to combat overfitting:

importance of reg. term

$$L = \frac{1}{M}\sum_i (\hat{y}^i - y^i)^2 + \lambda R(W)$$

Data loss

Regularization loss

L2 most popular: low loss when all weights are relatively small. More strongly penalizes large weights vs L1. Expresses preference for simple models (need large coefficients to fit a function to extreme outlier values).

Examples

L2 regularization: $R(W) = \sum_k \sum_l W_{k,l}^2$ (weight decay)

L1 regularization: $R(W) = \sum_k \sum_l |W_{k,l}|$

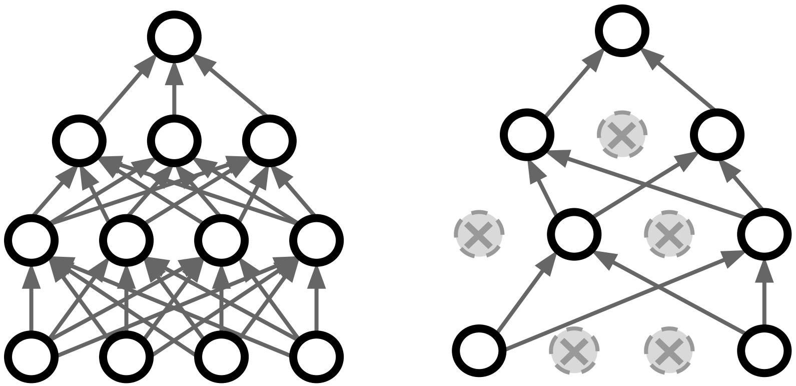
Elastic net (L1 + L2): $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

https://www.tensorflow.org/api_docs/python/tf/keras/regularizers

Next: implicit regularizers that do not add an explicit term; instead do something implicit in network to prevent it from fitting too well to training data

Design choices: regularization (dropout)

First example of an implicit regularizer.
During training, at each iteration of forward pass randomly set some neurons to zero (i.e., change network architecture such that paths to some neurons are removed).

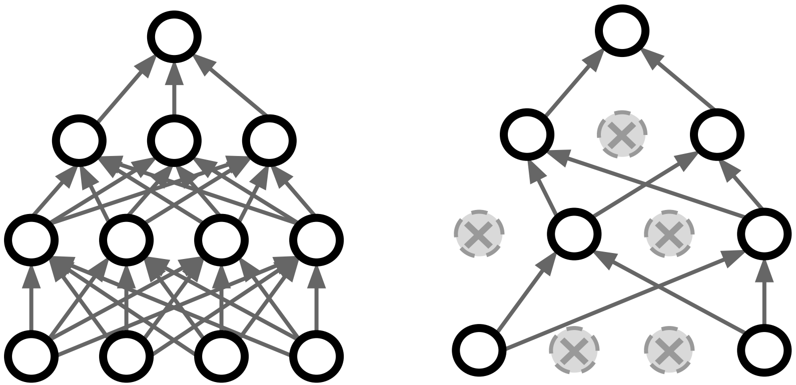


During testing, all neurons are active. But scale neuron outputs by dropout probability p, such that expected output during training and testing match.

Srivastava et al, “Dropout: A simple way to prevent neural networks from overfitting”, JMLR 2014. Figure credit: CS231n.

Design choices: regularization (dropout)

First example of an implicit regularizer.
During training, at each iteration of forward pass randomly set some neurons to zero (i.e., change network architecture such that paths to some neurons are removed).



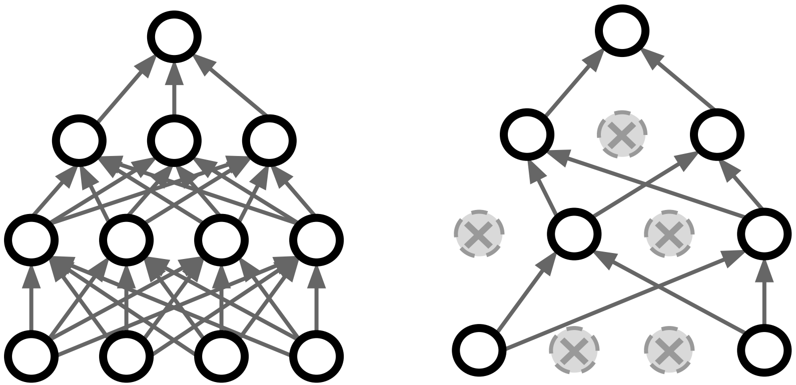
Probability of “dropping out” each neuron at a forward pass is hyperparameter p. 0.5 and 0.9 are common (high!).

During testing, all neurons are active. But scale neuron outputs by dropout probability p, such that expected output during training and testing match.

Srivastava et al, “Dropout: A simple way to prevent neural networks from overfitting”, JMLR 2014. Figure credit: CS231n.

Design choices: regularization (dropout)

First example of an implicit regularizer.
During training, at each iteration of forward pass randomly set some neurons to zero (i.e., change network architecture such that paths to some neurons are removed).



Probability of "dropping out" each neuron at a forward pass is hyperparameter p. 0.5 and 0.9 are common (high!).

Intuition: dropout is equivalent to training a large ensemble of different models that share parameters.

During testing, all neurons are active. But scale neuron outputs by dropout probability p, such that expected output during training and testing match.

Srivastava et al, "Dropout: A simple way to prevent neural networks from overfitting", JMLR 2014. Figure credit: CS231n.

Design choices: regularization (batch normalization)

Another example of an implicit regularizer.
Insert BN layers after FC or conv layers, before activation function.
During training, at each iteration of forward pass normalize neuron activations by mean and variance of minibatch. Also learn scale and shift parameter to get final output.

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

During testing, normalize by a fixed mean and variance computed from the entire training set. Use learned scale and shift parameters.

Design choices: regularization (batch normalization)

Another example of an implicit regularizer.
Insert BN layers after FC or conv layers, before activation function.
During training, at each iteration of forward pass normalize neuron activations by mean and variance of minibatch. Also learn scale and shift parameter to get final output.

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N} (x_{i,j} - \mu_j)^2$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Intuition: batch normalization allows keeping the weights in a healthy range. Also some randomness at training due to different effect from each minibatch sampling -> regularization!

During testing, normalize by a fixed mean and variance computed from the entire training set. Use learned scale and shift parameters.

Design choices: data augmentation

Augment effective training data size by simulating more diversity from existing data. Random combinations of:

- Translation and scaling
- Distortion
- Image color adjustment
- Etc.

Design choices: data augmentation

Augment effective training data size by simulating more diversity from existing data. Random combinations of:

- Translation and scaling
- Distortion
- Image color adjustment
- Etc.

Think about the domain of your data: what makes sense as realistic augmentation operations?

Deep learning for healthcare: the rise of medical data

Q: What are other examples of potential data augmentations?

(Raise hand or type in chat box)

Model inference

Maximizing test-time performance: apply data augmentation operations

Main idea: apply model on multiple variants of a data example, and then take average or max of scores

Can use data augmentation operations we saw during training! E.g.:

- Evaluate at different translations and scales
- Common approach for images: evaluate image crops at 4 corners and center, + horizontally flipped versions -> average 10 scores

Model ensembles

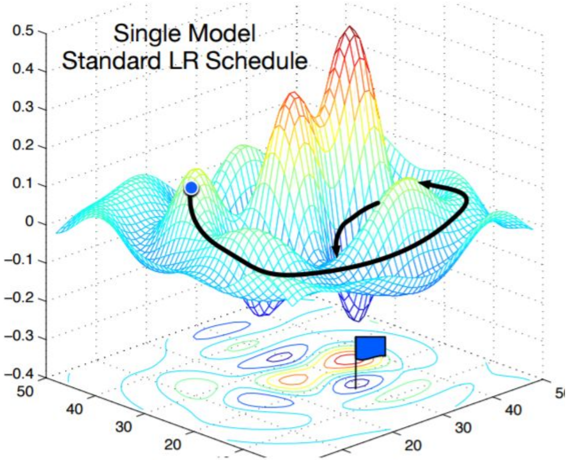
1. Train multiple independent models
2. At test time average their results

Enjoy 2% extra performance

Slide credit: CS231n

Model ensembles: tips and tricks

Instead of training independent models, use multiple snapshots of a single model during training!

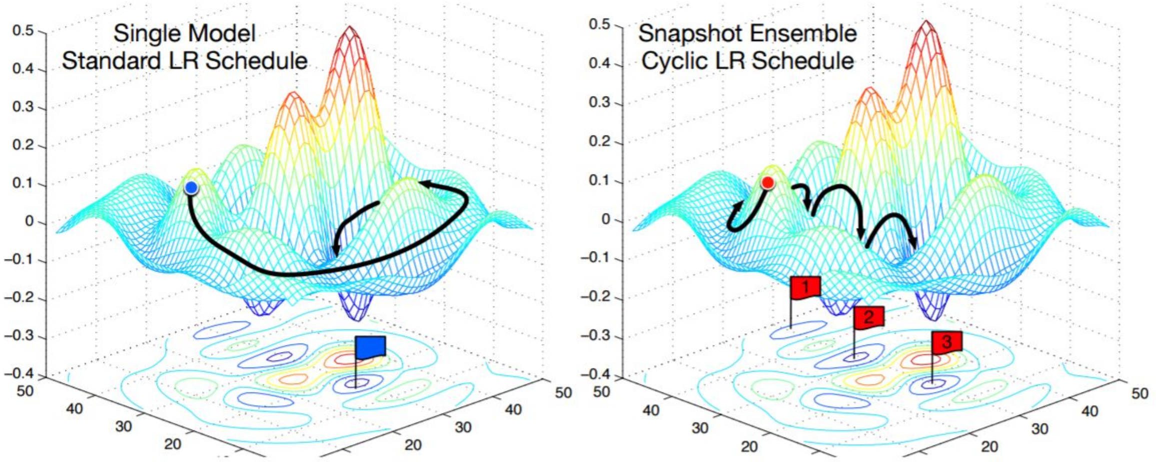


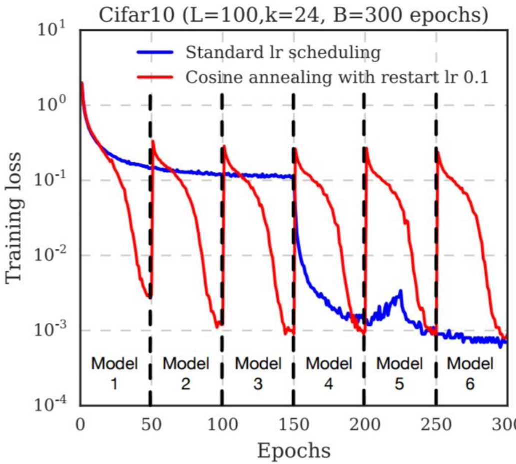
Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016
Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017

Slide credit: CS231n

Model ensembles: tips and tricks

Instead of training independent models, use multiple snapshots of a single model during training!





Cyclic learning rate schedules can make this work even better!

Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016
Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017

Slide credit: CS231n

Deeper models

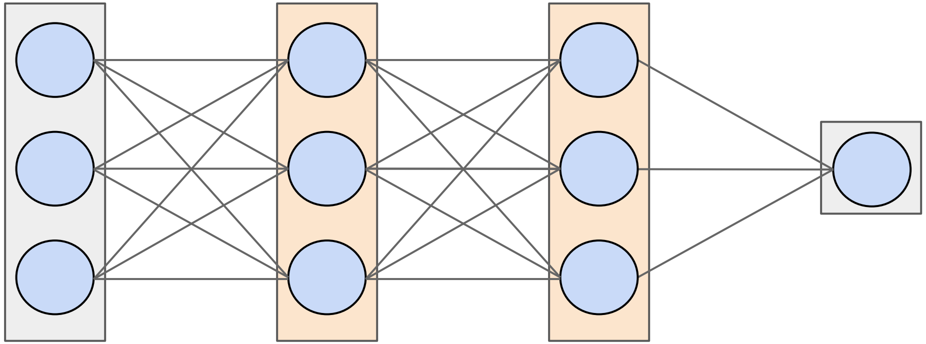
Training more complex neural networks is a straightforward extension

```
keras_model = tf.keras.models.Sequential([
    tf.keras.layers.Dense(units=3, activation='sigmoid', use_bias=True),
    tf.keras.layers.Dense(units=3, activation='sigmoid', use_bias=True),
    tf.keras.layers.Dense(units=3, activation='sigmoid', use_bias=True),
    tf.keras.layers.Dense(units=3, activation='sigmoid', use_bias=True),
    tf.keras.layers.Dense(units=3, activation='sigmoid', use_bias=True),
    tf.keras.layers.Dense(units=1, use_bias=True)
])
keras_model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=1e-2),
              loss='mse')
keras_model.fit(dataset, epochs=1000)
```

Now a 6-layer network

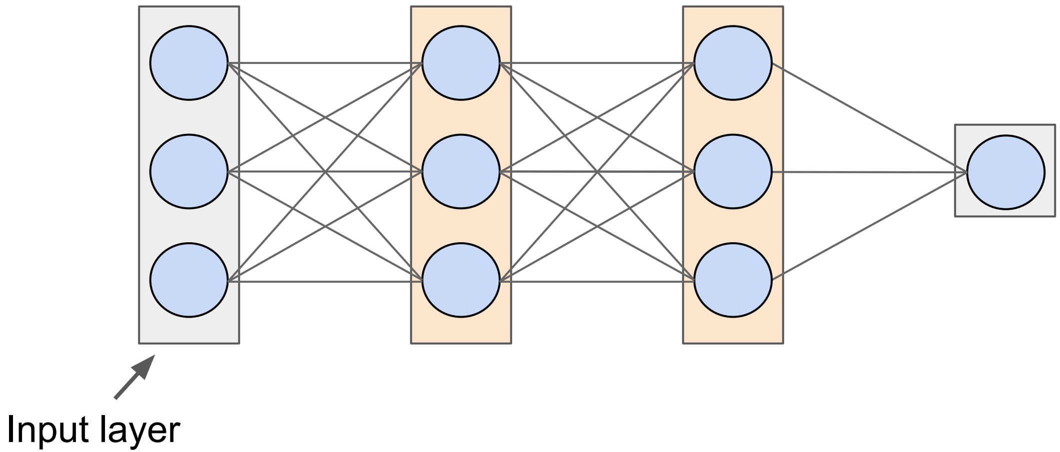
“Deep learning”

Can continue to stack more layers to get deeper models!



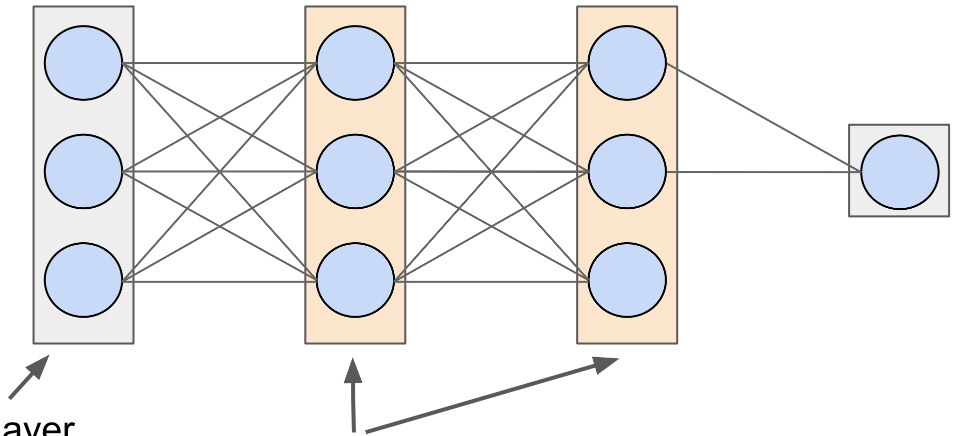
“Deep learning”

Can continue to stack more layers to get deeper models!



“Deep learning”

Can continue to stack more layers to get deeper models!

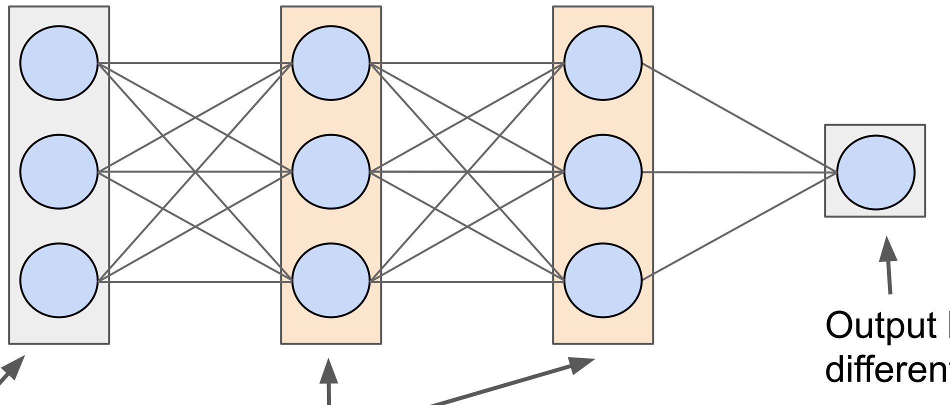


Input layer

“Hidden” layers - will see lots of diversity in size (# neurons), type (linear, convolutional, etc.), and activation function (sigmoid, ReLU, etc.)

“Deep learning”

Can continue to stack more layers to get deeper models!



Input layer

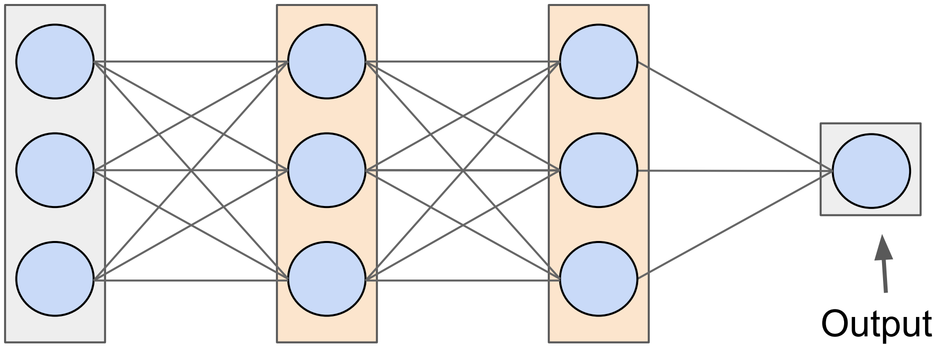
“Hidden” layers - will see lots of diversity in size (# neurons), type (linear, convolutional, etc.), and activation function (sigmoid, ReLU, etc.)

Output layer - will differ for different types of tasks (e.g. regression). Should match with loss function.

“Deep learning”

Vanilla fully-connected neural networks (MLPs) usually pretty shallow -- otherwise too many parameters! ~2-3 layers. Can have wide range in size of layers (16, 64, 256, 1000, etc.) depending on how much data you have.

Can continue to stack more layers to get deeper models!



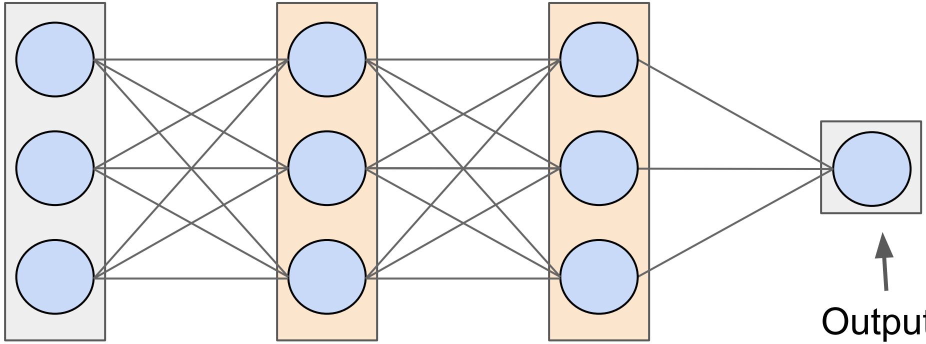
Input layer

“Hidden” layers - will see lots of diversity in size (# neurons), type (linear, convolutional, etc.), and activation function (sigmoid, ReLU, etc.)

Output layer - will differ for different types of tasks (e.g. regression). Should match with loss function.

“Deep learning”

Can continue to stack more layers to get deeper models!



Input layer

“Hidden” layers - will see lots of diversity in size (# neurons), type (linear, convolutional, etc.), and activation function (sigmoid, ReLU, etc.)

Output layer - will differ for different types of tasks (e.g. regression). Should match with loss function.

Vanilla fully-connected neural networks (MLPs) usually pretty shallow -- otherwise too many parameters! ~2-3 layers. Can have wide range in size of layers (16, 64, 256, 1000, etc.) depending on how much data you have.

Will see different classes of neural networks that leverage structure in data to reduce parameters + increase network depth

Deep learning

Q: What tasks other than regression are there?

(Raise hand or type in chat box)

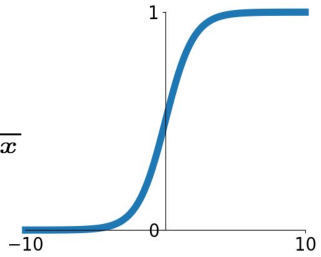
Common activation functions

You can find these in Keras:
https://keras.io/layers/advanced-activations/

You will see these extensively, typically after linear or convolutional layers.
They add nonlinearity to allow the model to express complex nonlinear functions.

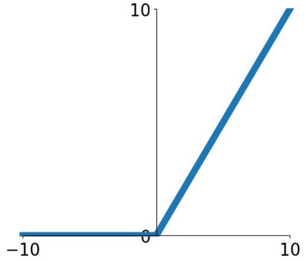
Sigmoid

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



ReLU

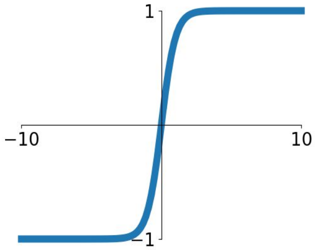
$\max(0, x)$



and many more...

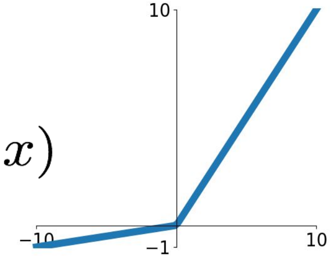
Tanh

$\tanh(x)$



Leaky ReLU

$\max(0.1x, x)$



Common activation functions

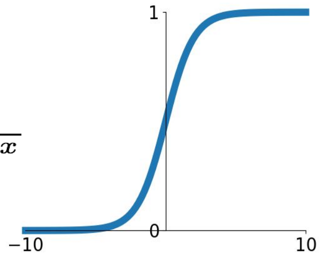
You can find these in Keras:
https://keras.io/layers/advanced-activations/

You will see these extensively, typically after linear or convolutional layers.
They add nonlinearity to allow the model to express complex nonlinear functions.

Typical in modern CNNs and MLPs

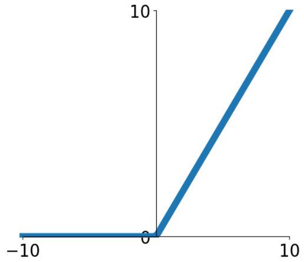
Sigmoid

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



ReLU

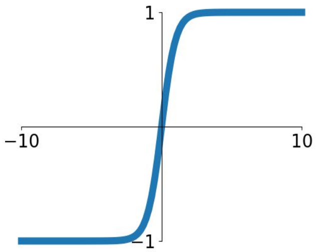
$$\max(0, x)$$



and many more...

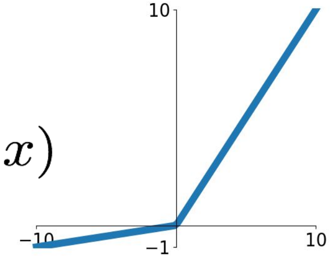
Tanh

$$\tanh(x)$$



Leaky ReLU

$$\max(0.1x, x)$$



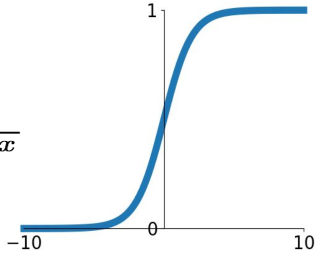
Common activation functions

You can find these in Keras:
https://keras.io/layers/advanced-activations/

You will see these extensively, typically after linear or convolutional layers.
They add nonlinearity to allow the model to express complex nonlinear functions.

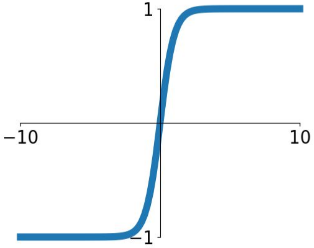
Sigmoid

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



Tanh

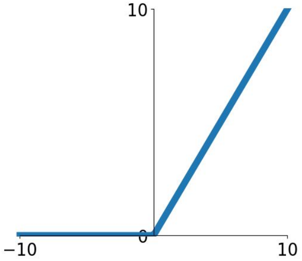
$$\tanh(x)$$



Will see in recurrent neural networks. Also used in early MLPs and CNNs.

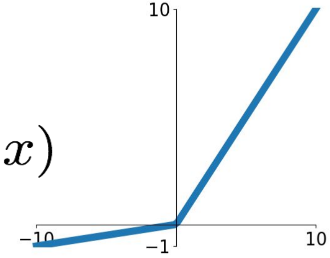
ReLU

$$\max(0, x)$$



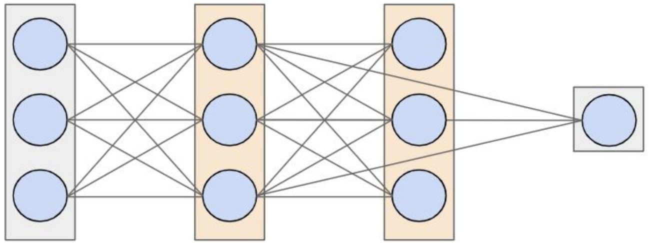
Leaky ReLU

$$\max(0.1x, x)$$

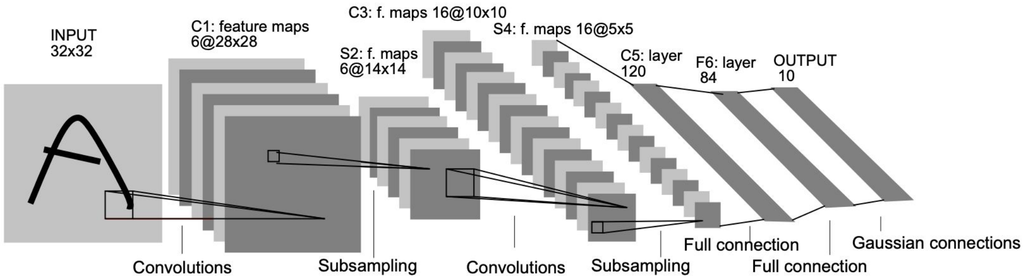


and many more...

Will see different classes of neural networks

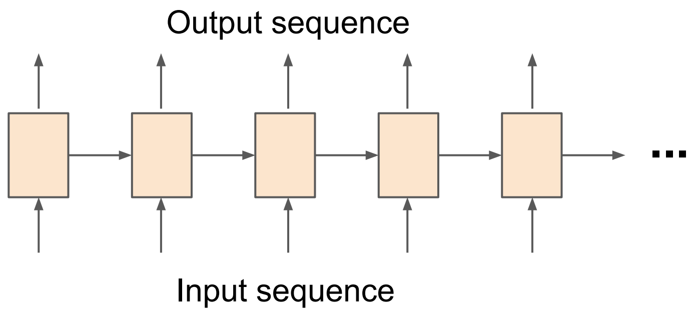


Fully connected neural networks
(linear layers, good for "feature vector" inputs)



Convolutional neural networks
(convolutional layers, good for image inputs)

Recurrent neural networks
(linear layers modeling recurrence relation across sequence, good for sequence inputs)



Project

Zoom Poll:

- Do you have a project in mind?
- Are you looking for project partners?

Summary

- Went over how to define, train, and tune a neural network
- This Friday's section (Zoom link on Canvas) will be a project partner finding section
- Next Friday's section will provide an in-depth tutorial on Tensorflow
- Next class: will go in-depth into
    - Medical image data
    - Classification models
    - Data, model, evaluation considerations