CSCI E-82a
Probabilistic Programming and AI
Lecture 13
Overview of Deep Learning

Steve Elston

HARVARD
Extension School

Building Blocks of Deep Learning

- Forward propagation and linear networks
- The perceptron
- Deep representations
- Nonlinearity and activation functions
- Learning with backpropagation
- Loss function
- Computing gradients with the chain rule
- Regularization for deep learning
- Optimization for deep learning

Function Approximation with Deep Neural Networks

- Deep neural networks are powerful **function approximators**

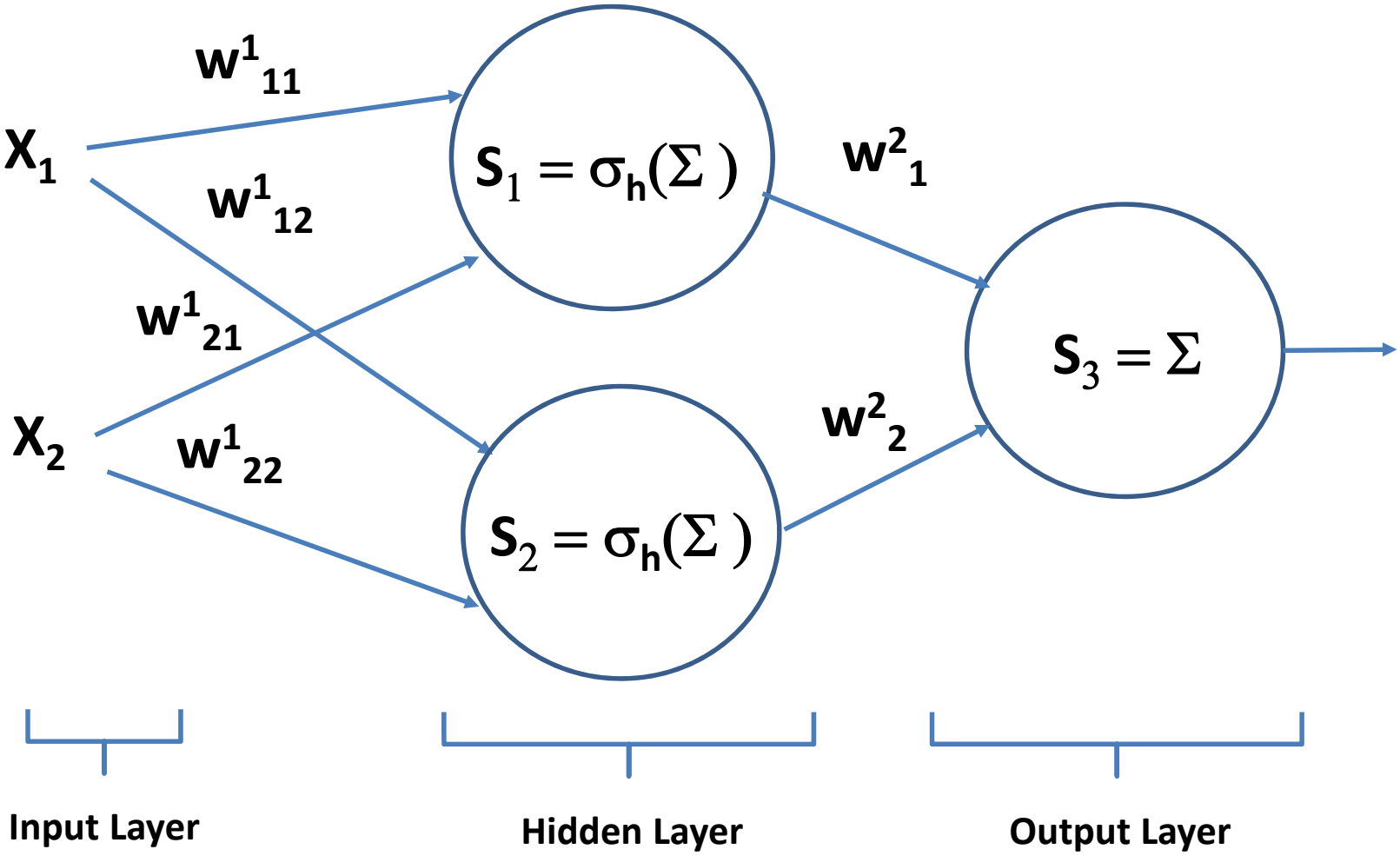
$$y = f(x)$$

  - X are the feature values
  - Y the labels
- Most deep neural networks use **supervised learning**
  - Labelled cases used to learn f(x)
  - f(x) is nonlinear and can be quite complex
  - Complexity leads to problems with generalization

A Better Deep Representation

- By mid-1980s need for architecture with **hidden layers** for **greater model capacity** was recognized
  - **Input layer**
  - Multiple **hidden layers**
  - **Output layer**
- Apply **nonlinear activations** in **hidden units**
- Can **fully connect** between layers
- **Learn weights** for complex function approximation
- Can solve XOR problem and much more

A Better Deep Representation



A Better Deep Representation

- What is the output of the simple network?
- Process called **forward propagation**
- Start with the output of the hidden layer:

$$S_1 = \sigma(\Sigma_i x_i * W^1_{1i})$$

$$S_2 = \sigma(\Sigma_i x_i * W^1_{2i})$$

- Next, compute the output of the output layer

$$S_3 = \Sigma_j W^2_j * \sigma(\Sigma_i x_i * W^1_{ji})$$

Model Capacity

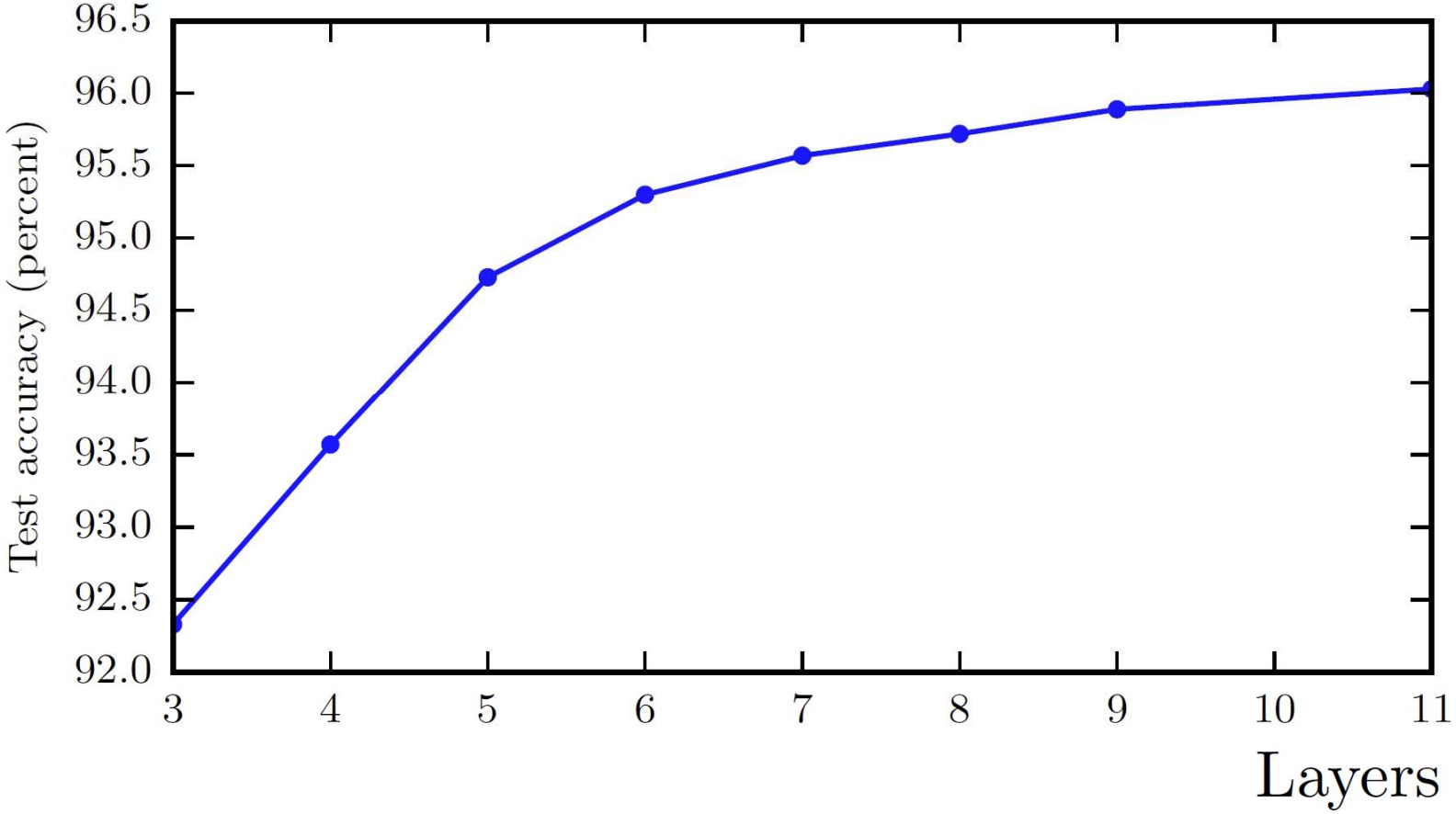
- The **universal approximation theorem**, Hornik (1991), tells us that an **infinitely wide hidden layer can represent any function**
- Usefulness limited:
  - It's nice to know we can represent complex functions
  - But, completely **infeasible** in practice
- What can we do?
  - Trade depth for breath

Model Capacity

- Model capacity is fundamentally related to the **bias-variance trade-off** of machine learning
  - **Low capacity** models have **high bias but low variance**
  - **High cap**acity models have **low bias but high variance**
- High capacity models have a tendency to be overfit
- We have more to say about this problem in another lesson

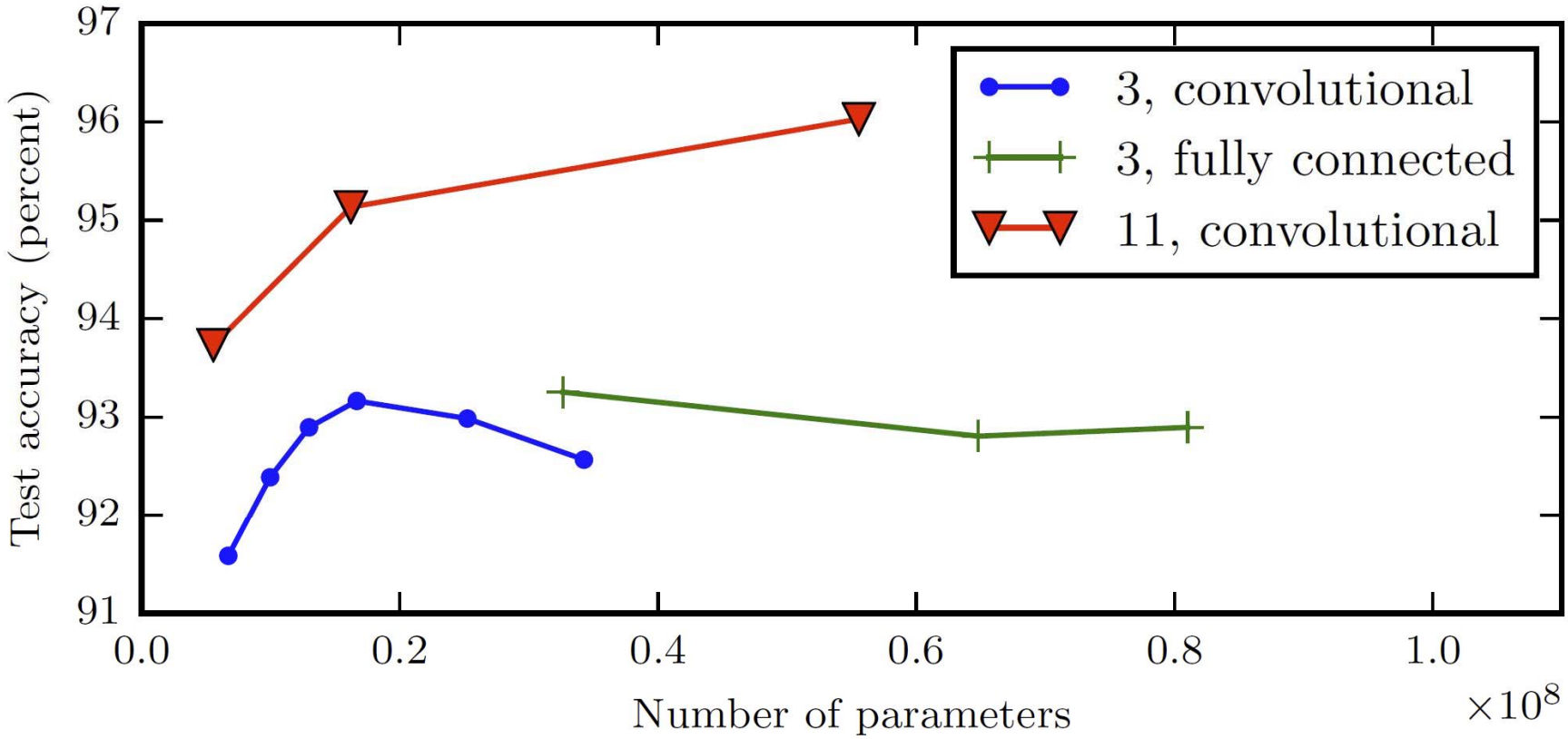
For a study of model capacity of extremely deep networks see He et. al.

Model Capacity



Model capacity with increasing depth. From Goodfellow et. al. 2014.

Model Capacity



Model capacity vs. number of parameters. From Goodfellow et. al. 2014.

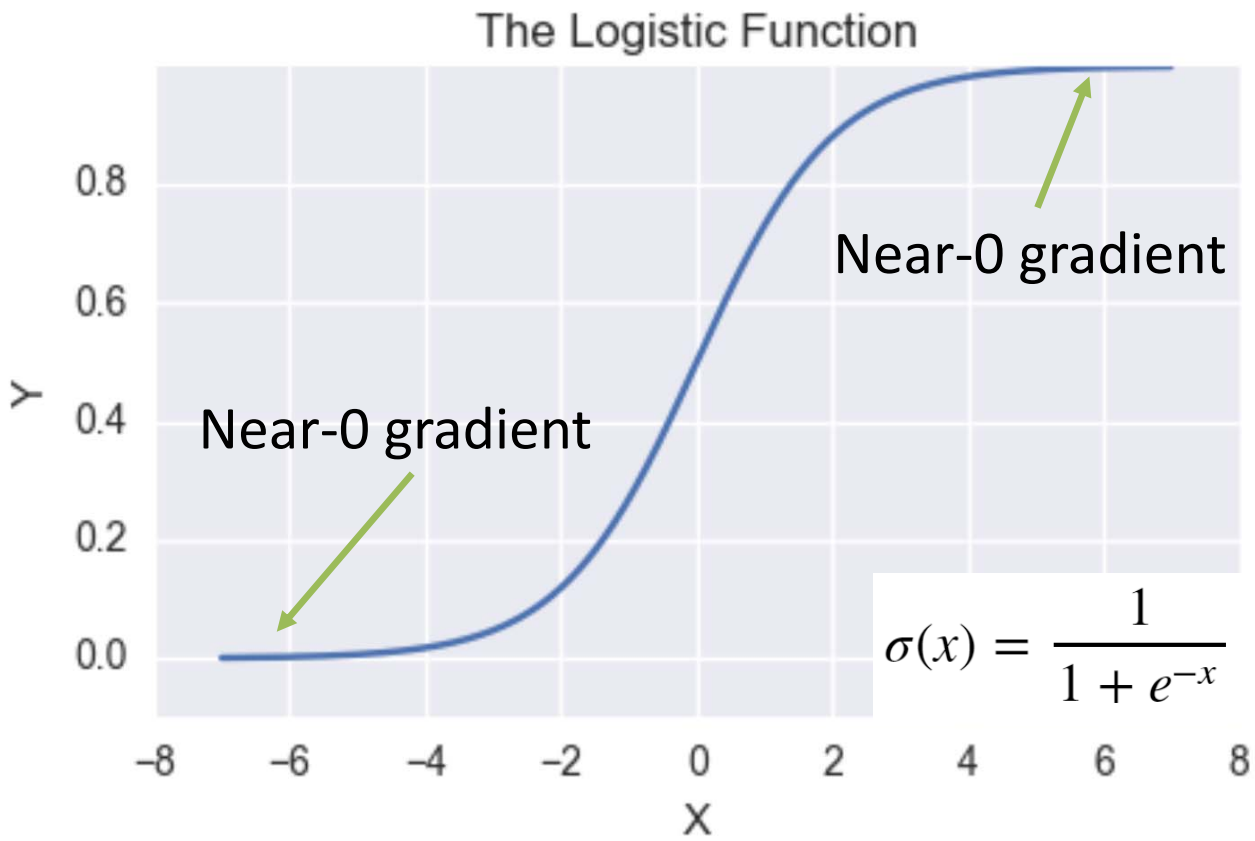
Activation functions

- Nonlinear activation is key to achieving good function approximation.
- Many activation functions have been tried, here are a few:

Function	How Used?	Comments
Sigmoid	Binary classifier output layer	Historically the most used

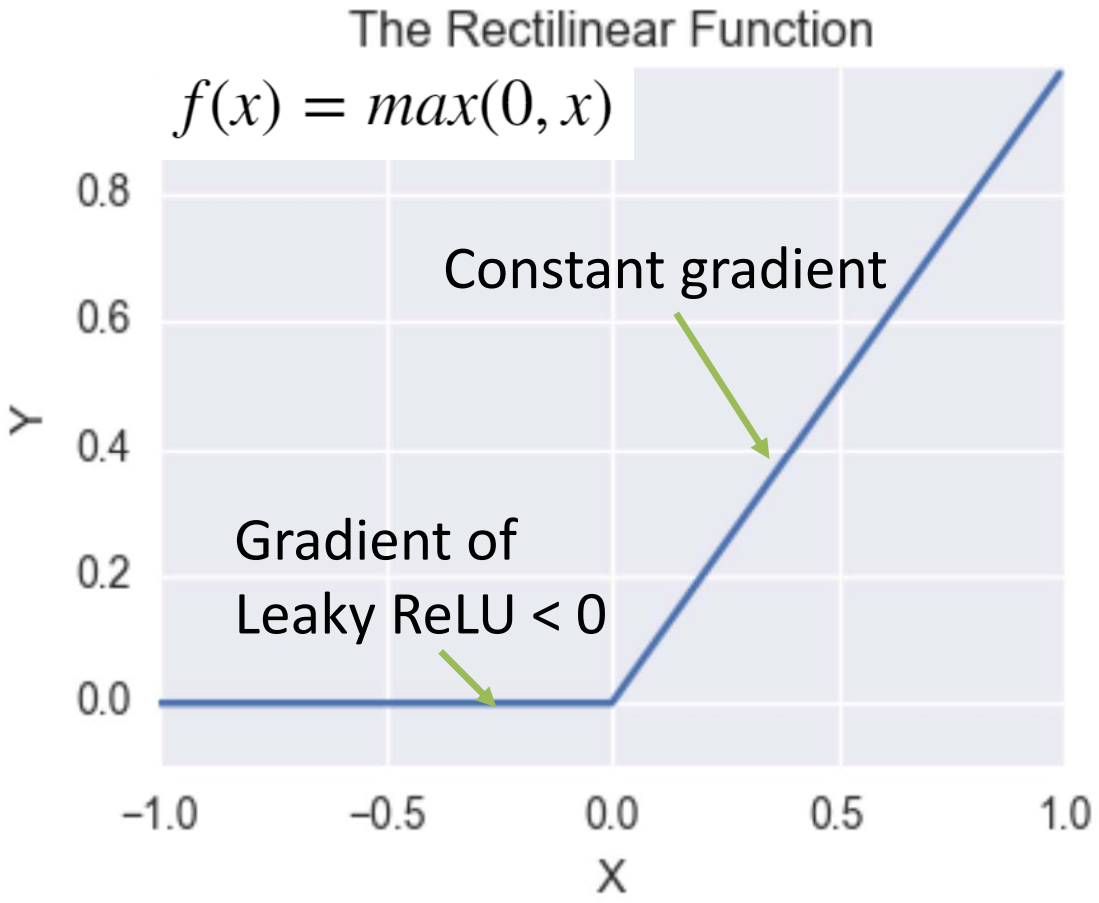
Activation functions

Sigmoid has vanishing gradients



Activation functions

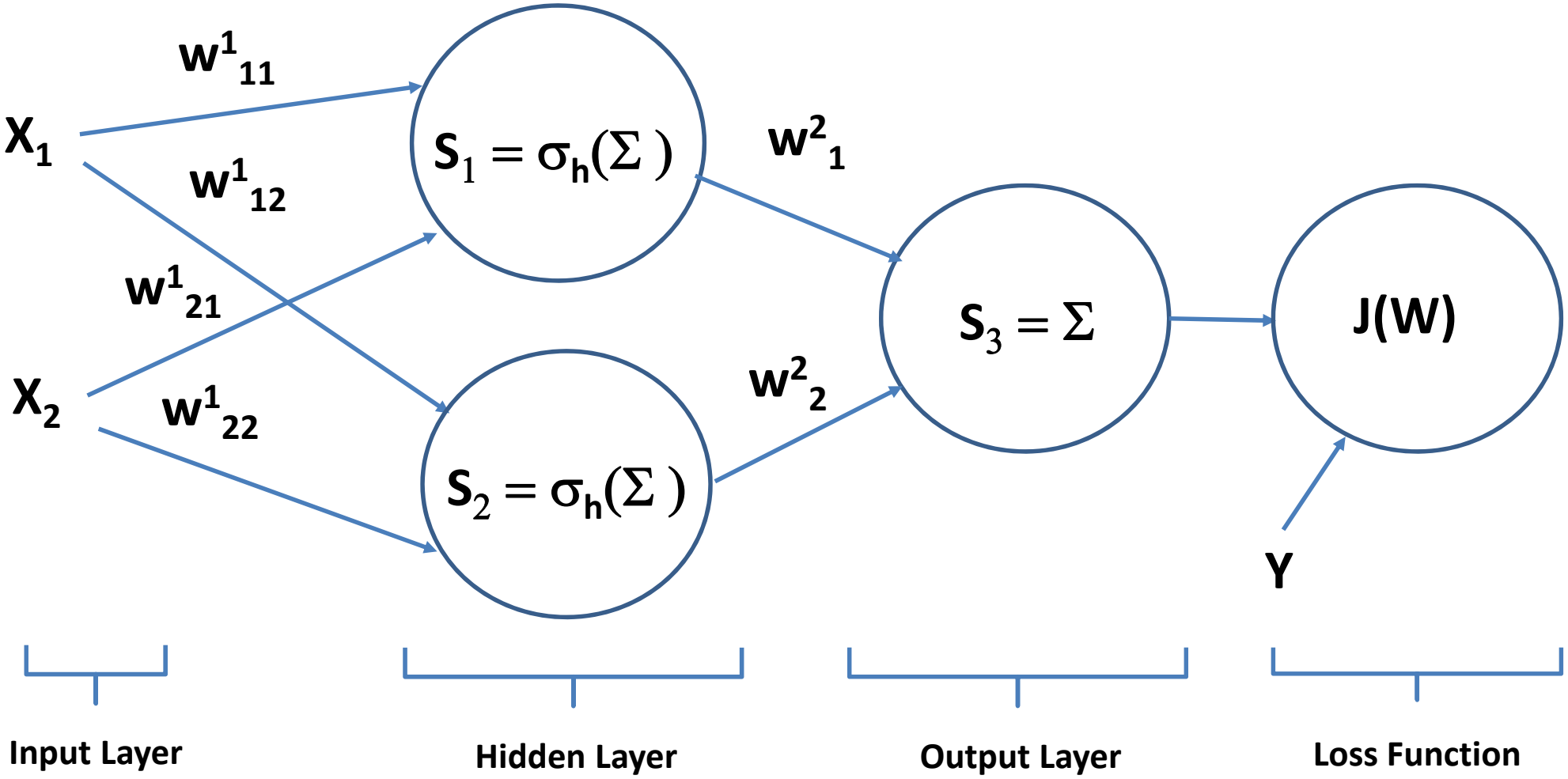
Rectilinear function has constant gradient for positive values



The Backpropagation Algorithm

- To find function approximation, f(x), we need to **learn model weights**
- The primary algorithm we use to learn model weights is known as **backpropagation**
  - Backpropagation was applied to learning (system identification) for control problems as early as 1960 by Henry Kelly and 1961 by Arthur Bryson for dynamic programming
  - First applied to neural networks by Paul Werbos in 1974
  - In 1986 by Rumelhart, Hinton and Williams showed that backpropagation was effective for learning the weights of hidden layers

The Backpropagation Algorithm



The Backpropagation Algorithm

To **learn model weight tensor** we must **minimize the loss function** using the **gradient:**

$$W_{t+1} = W_t + \alpha \nabla_W J(W_t)$$

Where**:**

W_t = the tensor of weights or model parameters at step t

$J(W)$ = loss function given the weights

$\nabla_W J(W)$ = gradient of J with respect to the weights W

α = step size or learning rate

The Back Propagation Algorithm

- Backpropagation is a **gradient decent algorithm**
- Weight updates are taken as small steps in the direction of the gradient of the loss function

$$\alpha \nabla_W J(W_t)$$

- Backpropagation converges when the gradient is approximately 0

Loss Functions for Training Neural Networks

- What are some choices for a loss function, $J(W)$, given the weight tensor?
- For regression problems use MSE
- Which loss function should we use for classification problems?
  - **Cross entropy** is a good choice, but is a bit abstract

Loss Functions for Training Neural Networks

What is **Shannon Entropy**?

$$\mathbb{H}(I) = E[I(X)]$$

Where: $E[X] =$ the expectation of X.
$I(X) =$ the information content of X.

But, we work with probability distributions, so:

$$\mathbb{H}(I) = E[-ln_b(P(X))] = -\sum_{i=1}^{n} P(x_i) ln_b(P(x_i)$$

Where: $P(X) =$ probability of X.
$b =$ base of the logarithm.

Loss Functions for Training Neural Networks

- We need to measure the difference between the distribution of our function approximation and the distribution of the data
- The **Kullback-Leibler divergence** between **two distributions P(X) and Q(X)** is such a measure**:**

$$\mathbb{D}_{KL}(P \parallel Q) = -\sum_{i=1}^{n} p(x_i)\, ln_b \frac{p(x_i)}{q(x_i)}$$

Loss Functions for Training Neural Networks

- How do we compute KL divergence?
- If we knew P(X) we would not need to compute KL divergence
- We can expand KL divergence as:

$$\mathbb{D}_{KL}(P \parallel Q) = \sum_{i=1}^{n} p(x_i)\, ln_b p(x_i) - \sum_{i=1}^{n} p(x_i)\, ln_b q(x_i)$$

$$\mathbb{D}_{KL}(P \parallel Q) = \mathbb{H}(P) + \mathbb{H}(P, Q)$$

$$\mathbb{D}_{KL}(P \parallel Q) = Entropy(P) + Cross\ Entropy(P, Q)$$

Loss Functions for Training Neural Networks

Given: $\mathbb{D}_{KL}(P \parallel Q) = \mathbb{H}(P) + \mathbb{H}(P, Q)$

The term $\mathbb{H}(P)$ is constant

So, we only need the **cross entropy** term:

$$\mathbb{H}(P, Q) = -\sum_{i=1}^{n} p(x_i)\, ln_b\, q(x_i)$$

Loss Functions for Training Neural Networks

How can we compute cross entropy when we don't know P(X):

$$\mathbb{H}(P, Q) = -\sum_{i=1}^{n} p(x_i)\, ln_b q(x_i)$$

Since we don't know P(X), use the approximation:

$$\mathbb{H}(P, Q) = -\frac{1}{N}\sum_{i=1}^{n} ln_b q(x_i)$$

The Chain Rule of Calculus

- In order to compute the gradients of the loss function though the layers of a deep neural network we need to apply the **chain rule of calculus**
- To consider a function z = f(y), where y = g(x); then z = f(g(x)). Then the derivative of z with respect to x is:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

The Chain Rule of Calculus

- We need the gradient of real-valued loss function, J, given a **M** dimensional weight tensor, W
- This leads to the general form of the chain rule:

$$\frac{\partial z}{\partial x} = \sum_{j \in M} \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

Or,

$$\nabla_x z = \left(\frac{\partial x}{\partial y} \right)^T \nabla_y z$$

Where, $\frac{\partial x}{\partial y}$ = is the nxm **Jacobian matrix** of partial derivatives

$\nabla_y z$ = the gradient of z with respect to y

Regularization for Deep Learning

- Deep learning models have very large numbers of parameters which must be learned.
  - Even with large training datasets there may only be a few samples per parameters
- Large number of parameters leads to high chance of **over-fitting** deep learning models
  - Over-fit models do not generalize
  - Over-fit models have poor response to input noise
- To prevent over-fitting we apply **regularization methods**

The Bias-Variance Trade-Off

- High capacity models fit training data well
  - Exhibit high variance
  - Do not generalize well; exhibit **brittle behavior**
  - $\text{Error}_{\text{training}} << \text{Error}_{\text{test}}$
- Low capacity models have high bias
  - Generalize well
  - Do not fit data well
- Regularization adds bias
  - Strong regularization adds significant bias
  - Weak regularization leads to high variance

The Bias-Variance Trade-Off

- How can we understand the bias-variance trade-off?
- We start with the error:

$$\Delta y = E\left[Y - \hat{f}(X)\right]$$

Where:

$Y =$ the label vector.

$X =$ the feature matrix.

$\hat{f}(x) =$ the trained model.

The Bias-Variance Trade-Off

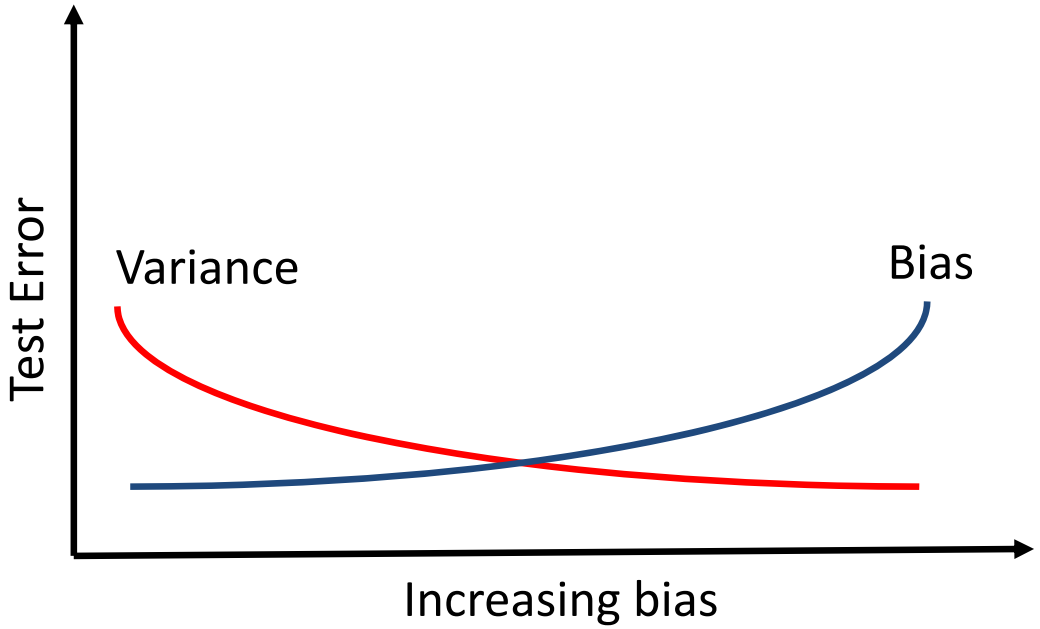
- We can expand the error term

$$\Delta x = \left(E[\hat{f}(X)] - \hat{f}(X)\right)^2 + E\left[(\hat{f}(X) - E[\hat{f}(X)])^2\right] + \sigma^2$$

$$\Delta x = Bias^2 + Variance + Irreducible\ Error$$

- Increasing bias decreases variance
- Notice that even if the bias and variance are 0 there is still irreducible error

The Bias-Variance Trade-Off



l2 Regularization

- Over-fit models tend to have parameters (weights) with extreme values
- One way to regularize models is to limit the values of the parameters
- We add a small bias term to (greatly) reduce the variance

l2 Regularization

- One way to limit the size of the model parameters is to constrain the **l2** or **Euclidian norm**:

$$||W||^2 = \left(w_1^2 + w_2^2 + \ldots + w_n^2\right)^{\frac{1}{2}} = \left(\sum_{i=1}^{n} w_i^2\right)^{\frac{1}{2}}$$

- The regularized loss function is then:

$$J(W) = J_{MLE}(W) + \lambda||W||^2$$

- Where λ is the regularization hyperparameter
  - Large λ increases bias but reduces variance
  - Small λ decreases bias and increases variance

l2 Regularization

- l2 regularization goes by many names
- Is called Euclidian norm regularization
- First published by Andrey Tikhonov regularization, in late 1940s
  - Only published in English in 1977
  - Is known as **Tikhonov regularization**
- In the statistics literature the method is often called **ridge regression**
- In the engineering literature is referred to as **pre-whitening**

l1 Regularization

- Regularization can be performed with other norms
- The **l1 (min-max) norm** is another common choice
- Conceptually, l1 norm limits the sum of the absolute values of the weights:

$$\|W\|^1 = \left(|w_1| + |w_2| + \ldots + |w_n|\right) = \left(\sum_{i=1}^{n} |w_i|\right)^1$$

- The l1 norm is also known as the **Manhattan distance** or **taxi cab distance**, since it is the distance traveled on a grid between two points.

l1 Regularization

- Given the l1 norm of the weights, the loss function becomes:

$$J(W) = J_{MLE}(W) + \alpha ||W||^1$$

- Where α is the regularization hyperparameter
  - Large α increases bias but reduces variance
  - Small α decreases bias and increases variance
- The l1 constraint drives some weights to exactly 0
  - This behavior leads to the term **lasso regularization**

Early Stopping

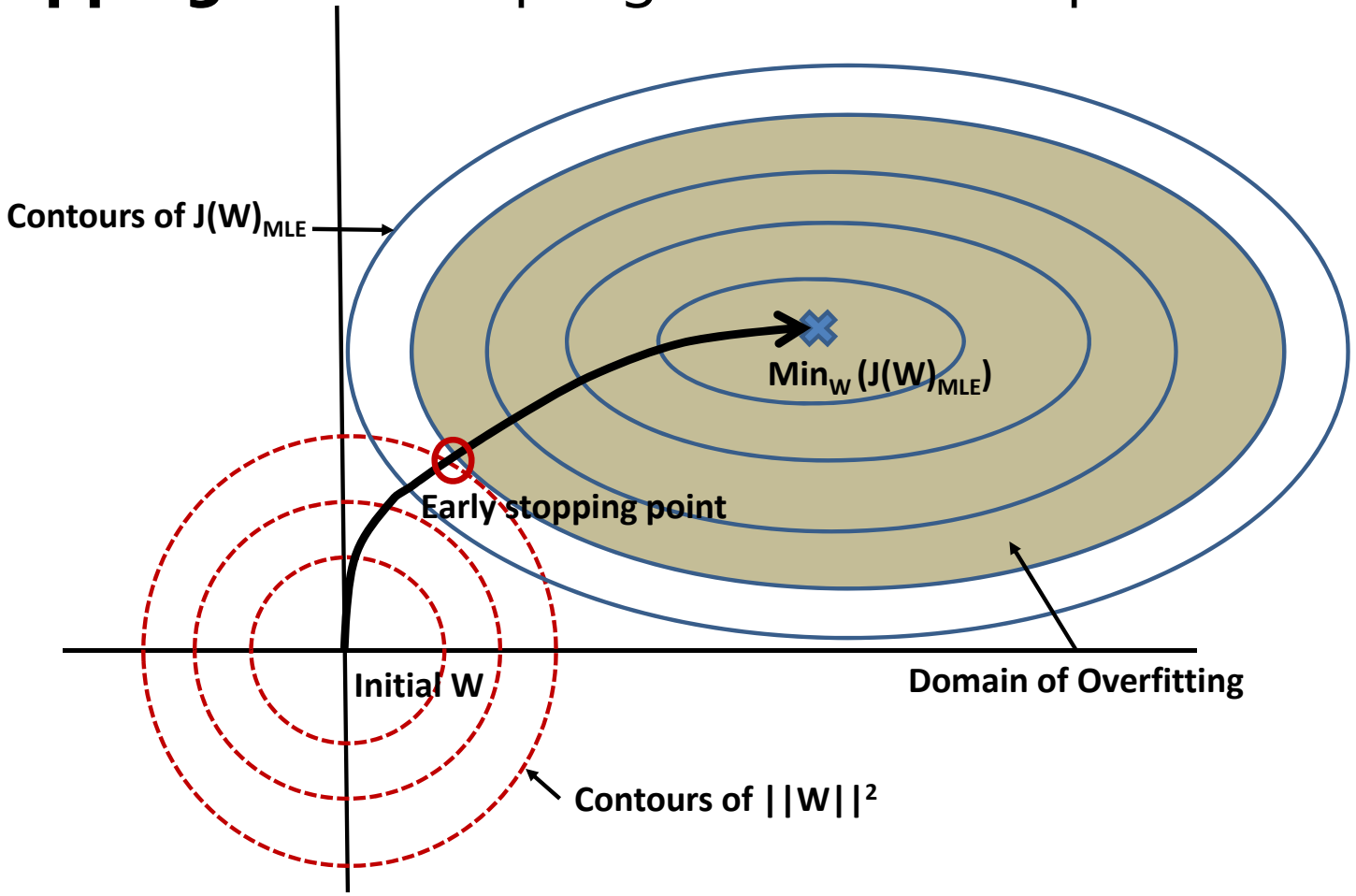
- **Early stopping** is an old and simple idea
- Stop updating the model weights before the model becomes overfit
- Early stopping is analogous to l2 regularization
- We can formulate the regularized loss function as:

$$argmin_W J(W) = J(W)_{MLE} + \alpha ||W||^2$$

- Where α is the regularization hyperparameter

Early Stopping

Early stopping has a simple geometric interpretation



Dropout regularization

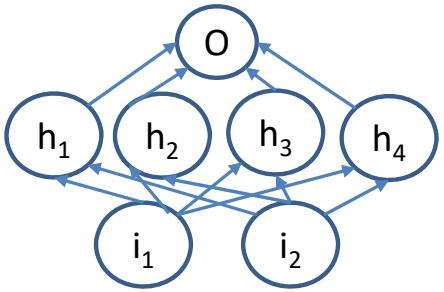
- Overfit deep network models tend to suffer from a problem of co-adaptation
  - With limited training data weight tensors become adapted to the training data
  - Such a model is unlikely to generalize
- We need a way to break the co-adaptation of the weight tensor

Dropout regularization

- **Dropout regularization** is a conceptually simple method unique to deep learning
  - At each step of the gradient decent some **fraction, p, of the weights are dropped-out** of each layer
  - The result is a series of models trained for each dropout sample
  - The final model is a **geometric mean** of the individual models
- Weight values are clipped in a small range as a further regularization
- For full details see the readable paper by Srivastava et. al., 2014 http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

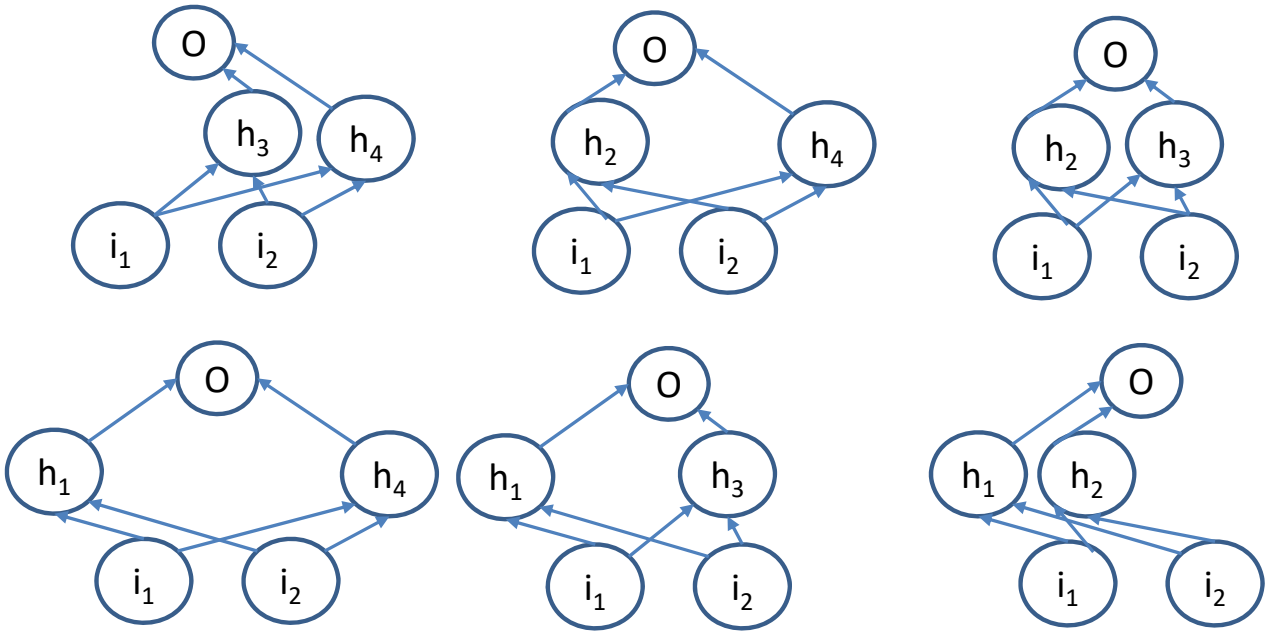
Dropout regularization

- Let's look at a simple example of a network with one hidden layer:



Dropout regularization

- For p = 0.5 here are six of the possible samples:



Batch Normalization

- In deep neural networks there is a high chance that units in a hidden layer have a **large range of output values**
  - Causes shifts in the covariance of the output values
  - Leads to difficulty computing the gradient
  - Slows convergence
- A solution is to normalize the output of the hidden layers in the network as a batch
- This simple idea can be really effective
- For more details see Sergey and Szegedy, 2015: https://arxiv.org/pdf/1502.03167.pdf

Optimization for Deep Neural Networks

- Training deep neural networks requires learning a large number parameters
- Parameters are learned by gradient descent
  - Is an optimization method
  - Must be executed on a large scale
- The optimization problem is ill-posed
  - Can have slow convergence
  - May not have unique solution
  - But, a good solution is often good enough

Optimization for Deep Neural Networks

- Neural networks **learn weights** using the backpropagation algorithm
- Weights are learned using the **gradient descent** method:

$$W_{t+1} = W_t + \alpha \nabla_W J(W_t)$$

Where**:**

W_t = the tensor of weights or model parameters at step t

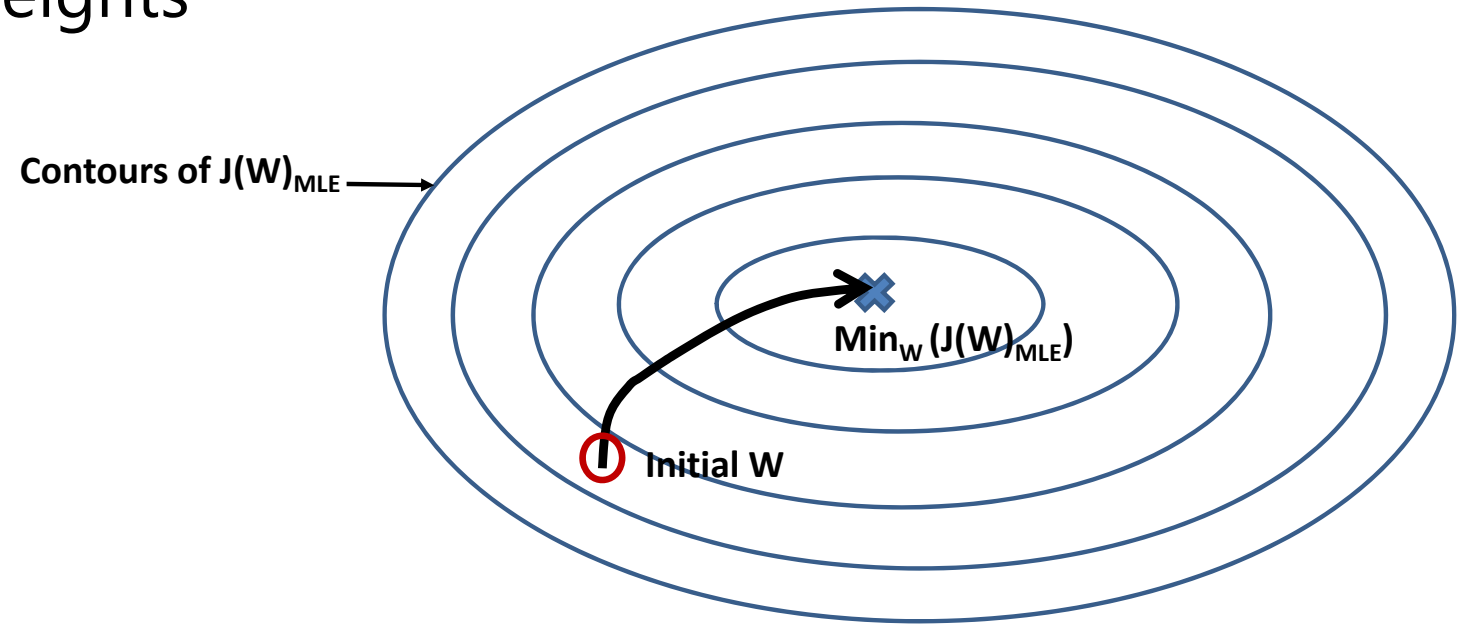
$J(W)$ = loss function given the weights

$\nabla_W J(W)$ = gradient of J with respect to the weights W

α = step size or learning rate

Local Convergence of Gradient Descent

- Ideally, the loss function, J(W), is **convex** with respect to the weights



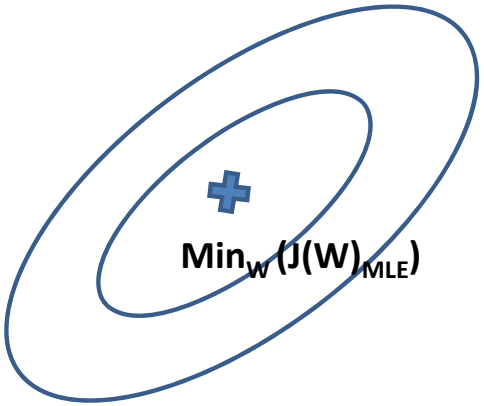
- Convex loss function has **one unique minimum**
- **Convergence** for convex loss function is **guaranteed**

Local Convergence of Gradient Descent

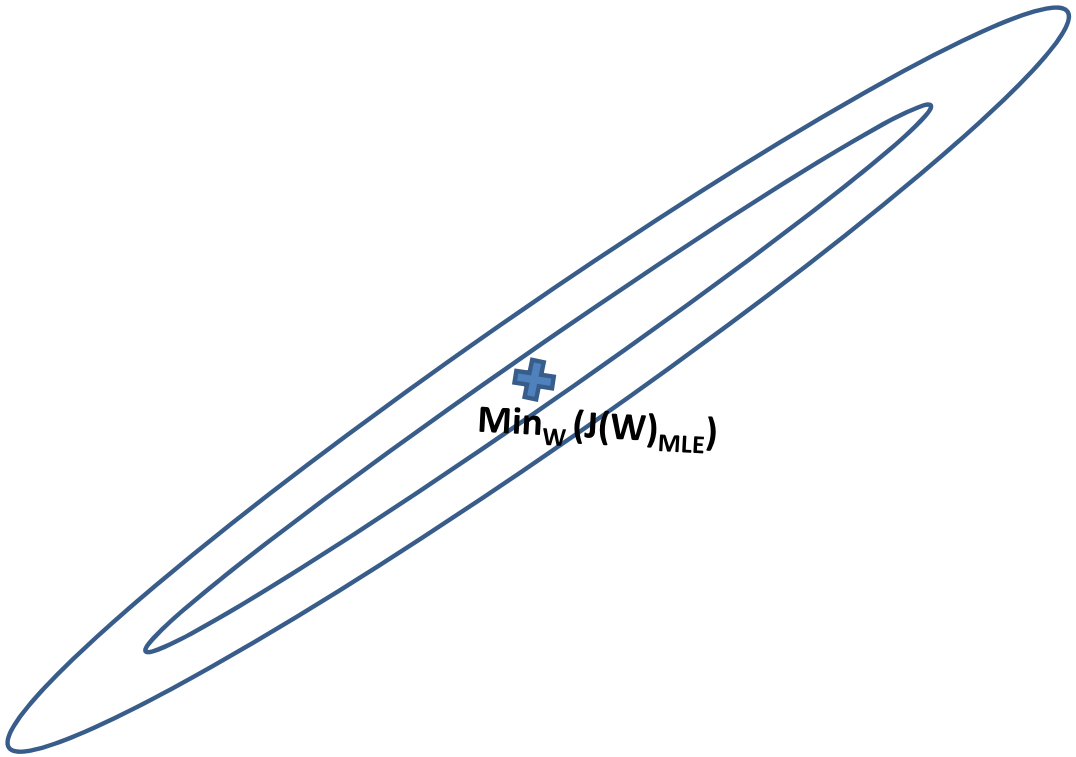
- Real-world loss functions are **typically not convex**
- There can be multiple minimums and maximums; a **multi-modal loss function**
- Finding the **globally optimal solution** is hard!
- The minimum reached by an optimizer depends on the **starting value of W**
- In practice, we are happy with a **good local solution**, if not, the globally optimal solution
- First order optimization found to perform as well, or better, than second order

The Nature of Gradients

- Example of well-conditioned and ill-conditioned gradients:



well-conditioned gradient



ill-conditioned gradient

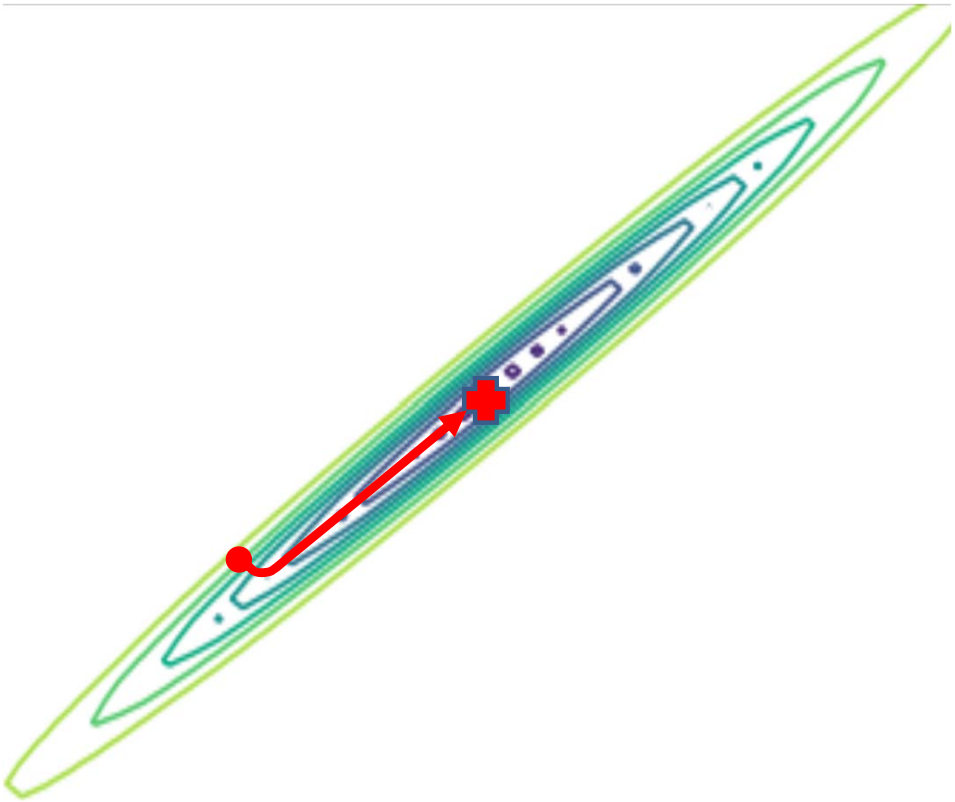
Vanishing and Exploding Gradient Problems

- There is no guarantee that the gradient of the loss function is well behaved
- The gradient can **vanish**
  - Flat spots in the gradient
  - Imagine a loss function with a long narrow valley
  - Slow convergence
- The gradient can **explode**
  - Sudden changes in the gradient; falling off a cliff!
  - Very large step; optimizer over-shoots the minimum point

Vanishing and Exploding Gradient Problems

- What can be done about extreme gradient problems?
- Dealing with vanishing gradient can be difficult
  - Normalization of input values
  - Regularization is essential!
- Dealing with exploding gradients is easy
  - **Gradient clipping** prevents extreme values

Convex vs. Non-Convex Optimization

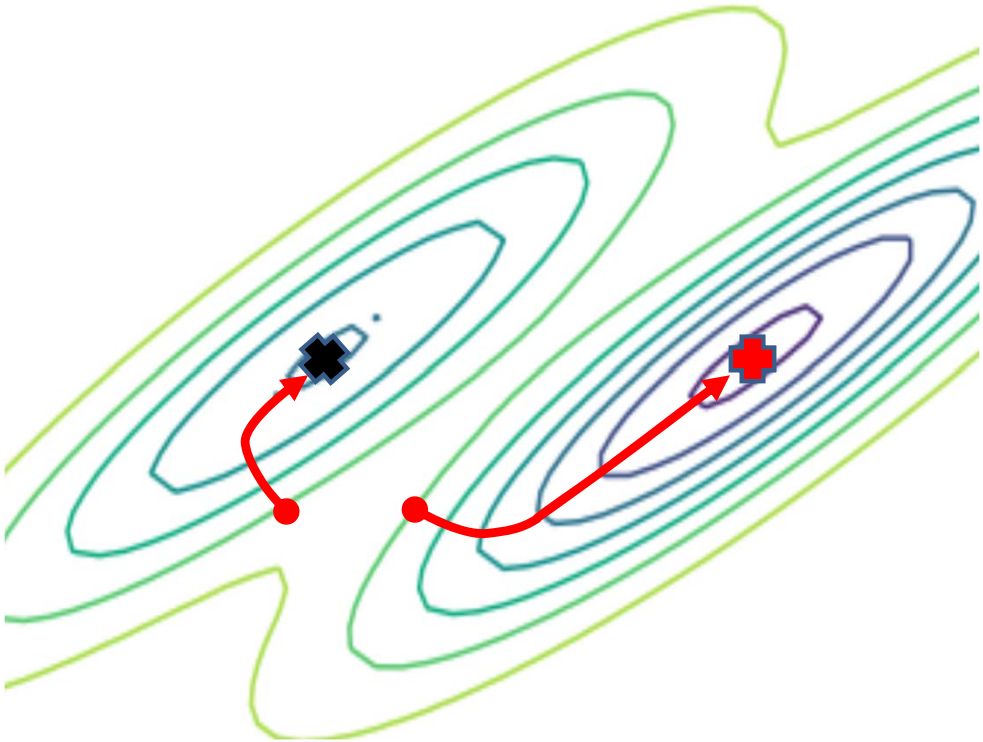


Convex loss
Poorly conditioned

Gradient descent is well-behaved with **convex loss function**

- Only 1 **global minimum**
- From any starting point the gradient leads to global minimum

Convex vs. Non-Convex Optimization

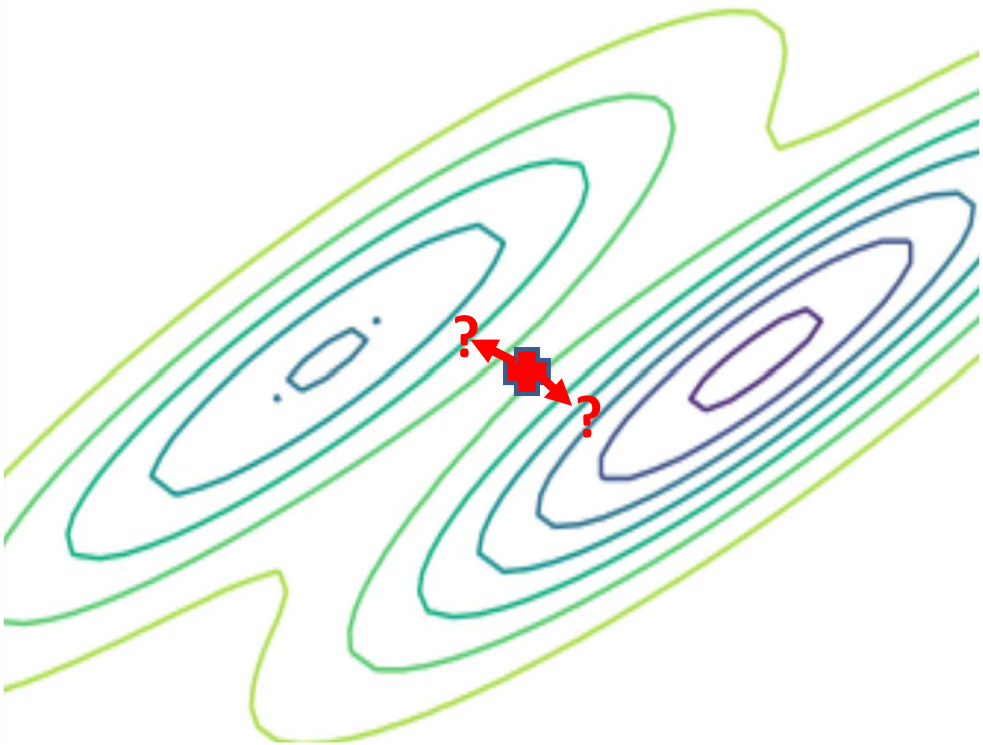


Nonconvex loss

Gradient descent can be problematic with **nonconvex loss function**

- There is a **global minimum**
- Possibly many **local minimums**
- Minimum found with gradient descent depends on starting point
- A good minimum may be good enough

Convex vs. Non-Convex Optimization



Nonconvex loss

Gradient descent can be problematic with **nonconvex loss function**

- Can get stuck at **saddle point!**
- Gradient is ambiguous at saddle point

Stochastic Gradient Descent

- We need a more scalable way to apply gradient descent
- **Stochastic gradient descent** is just such a method
- The weight tensor update for stochastic gradient descent follows this relationship:

$$W_{t+1} = W_t + \alpha \, E_{\hat{p}data}\left[\nabla_W J(W_t)\right]$$

Where:

$\hat{p}data$ is the Bernoulli sampled mini-batch

$E_{\hat{p}data}[]$ is the expected value of the gradient given the Bernoulli sample

Stochastic Gradient Descent

- Stochastic gradient descent is known to converge well in practice
- Empirically, using mini-batch samples provide a better exploration of the loss function space
  - Can help solution escape from small local gradient problems
  - Sampling is dependent on mini-batch size

Stochastic Gradient Descent

- Stochastic gradient descent algorithm

```
Random_sort(cases)
while(grad > stopping_criteria):
    mini-batch = sample_next_n(cases)
    grad = compute_expected_grad(mini_batch)
    weights = update_weights(weights, grad)
```

- Notice that the addition rounds repeat the samples
  - In practice this does not create much bias
  - For large samples this may not happen

Stochastic Gradient Descent with Momentum

- The stochastic gradient descent algorithm can be slow to converge if flat spots in the gradient are encountered
- What is a solution?
- Add **momentum** to the gradient; $momentum = m \cdot v$
  - Analogy with Newtonian mechanics;
  - Where:

m is the mass
v is the velocity

Stochastic Gradient Descent with Momentum

Letting the mass be 1.0 update of the weight tensor is:

$$v^{(l)} = momentum \cdot v^{(l-1)} + lr \cdot \nabla_W J(W^{(l)})$$

$$W^{(l+1)} = W^{(l)} + v^{(l)}$$

Where:

$v^{(l)}$ is the velocity at step l
$momentum$ is the momentum multiplier
lr is the learning rate

Notice there are now two hyperparameters

Adaptive Stochastic Gradient Descent

- A single learning rate is not likely to be optimal
  - Far from the minimum, a large learning rate speeds convergence
  - Near the minimum a small learning rate presents over-shooting the minimum
- What can improve the convergence
- Use a manually created learning schedule
  - Introduces additional hyperparameters
- Use an adaptive algorithm
  - Learning rate is adjusted based on the estimates of the gradient

Selecting Initial Weight Values

- To prevent weights from becoming linearly dependent the **initial values must be randomly selected**
  - Otherwise, some weight values are never learned
- Simple truncated Gaussian or Uniform distributed values work well in practice
  - This process is referred as adding **fuzz** to the initial values