Neural Networks

Lecture 2 - Working with Neural Networks

Igor Janos

Agenda

1. Back-propagation Algorithm with Neural Network
2. Machine Learning Project
    - Goals, Metrics, Datasets, Splits, Models
    - Model-centric, Data-centric Approach
3. NN Training Techniques
4. Optimizers

1 Back-propagation with NN

Example - Logistic Regression

Backward Pass

$$\mathcal{X} = \{(x_1 = 0.1, x_2 = 0.7) \rightarrow (y = 1)\}$$

$x = \begin{bmatrix} 0.1 \\ 0.7 \end{bmatrix}$

$z = 0.32 \longrightarrow a = 0.5793 \longrightarrow \mathcal{L} = 0.5459$

$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$

$$z(x_1, x_2, w_1, w_2, b) = z = w_1 x_1 + w_2 x_2 + b \rightarrow a(z) = a = \sigma(z) \rightarrow \mathcal{L}(a, y) = \mathcal{L} = -(y \log a + (1 - y) \log(1 - a))$$

$w = \begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix} \quad b = 0.1$

$w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \quad b$

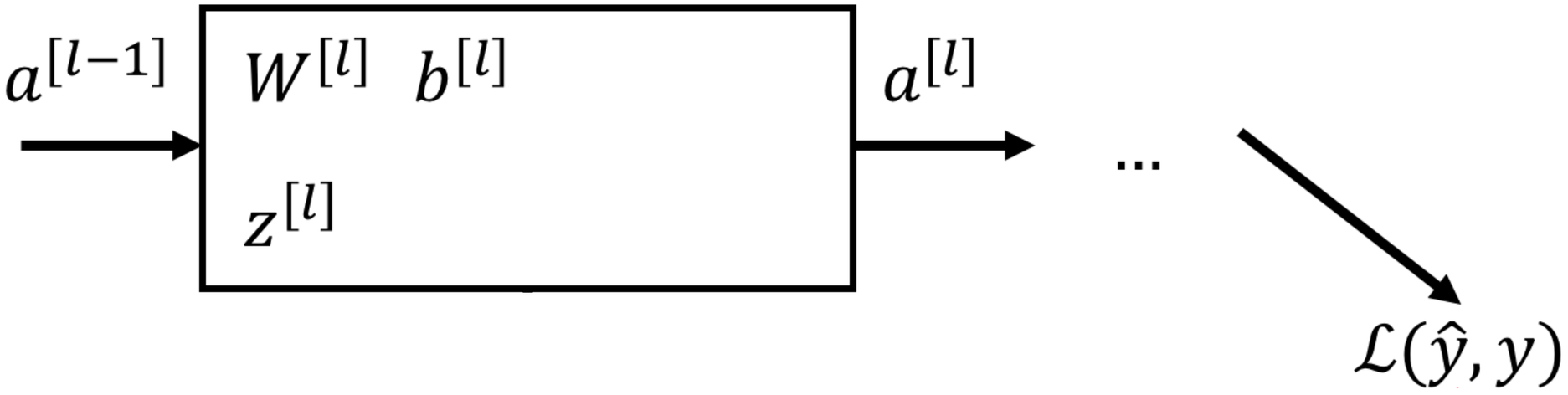
$$\nabla w = \begin{bmatrix} -0.04207 \\ -0.29447 \end{bmatrix} \qquad \nabla b = -0.4207$$

Back-propagation with Neural Network

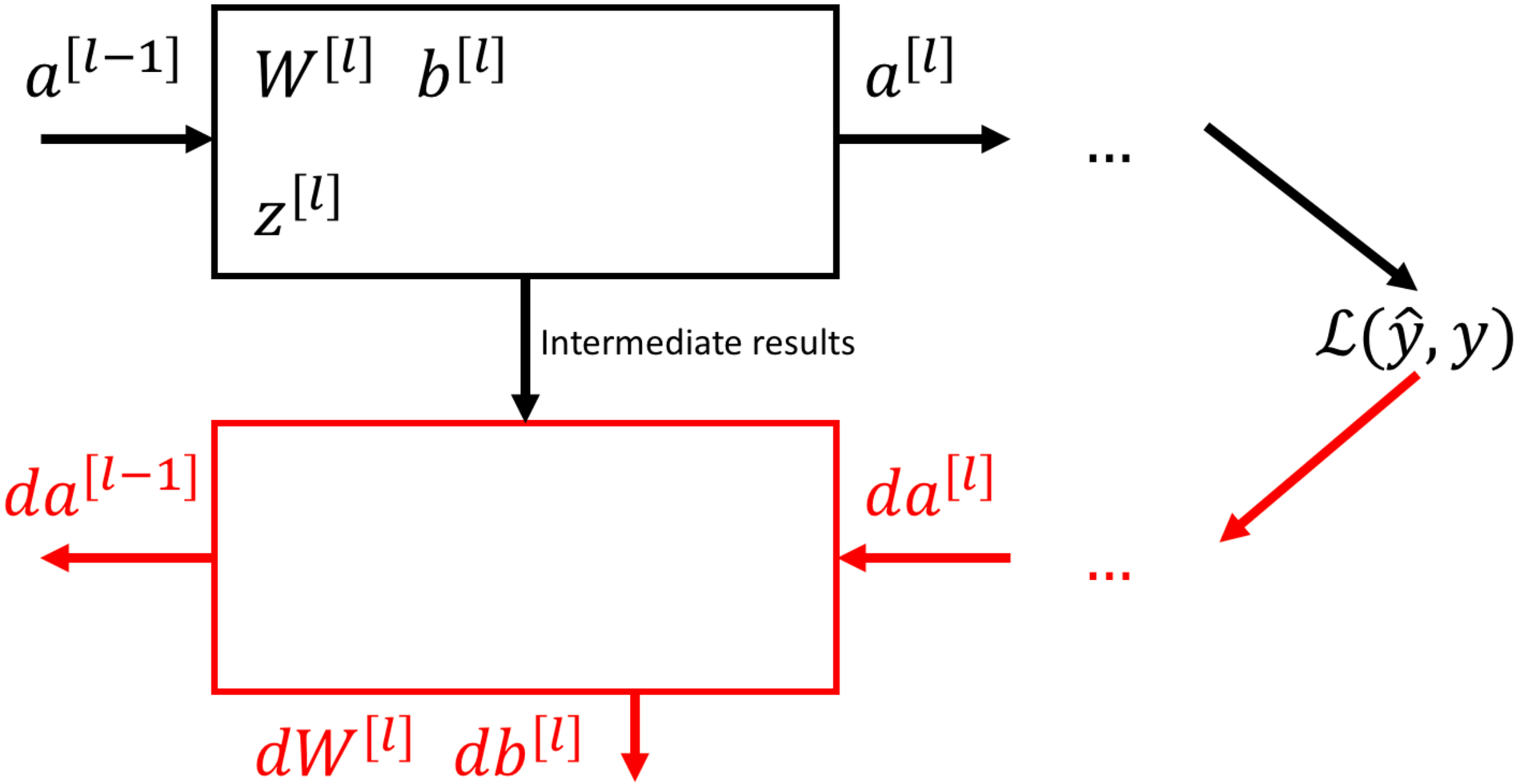
- Simplify notation

$$\frac{\partial \mathcal{L}(\hat{y}, y)}{\partial W^{[1]}} = \text{“dW1”}$$

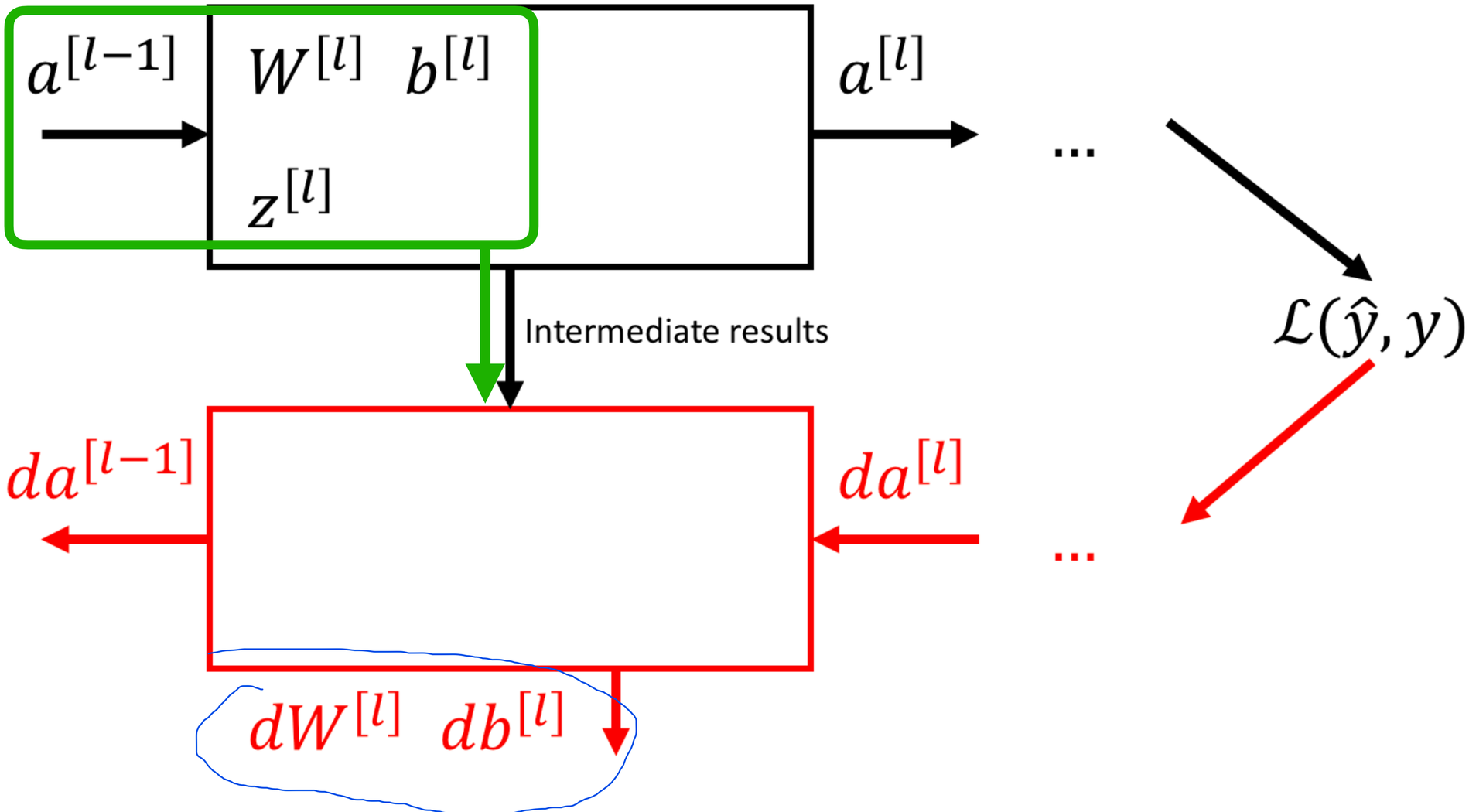
Back-propagation with Neural Network



Back-propagation with Neural Network

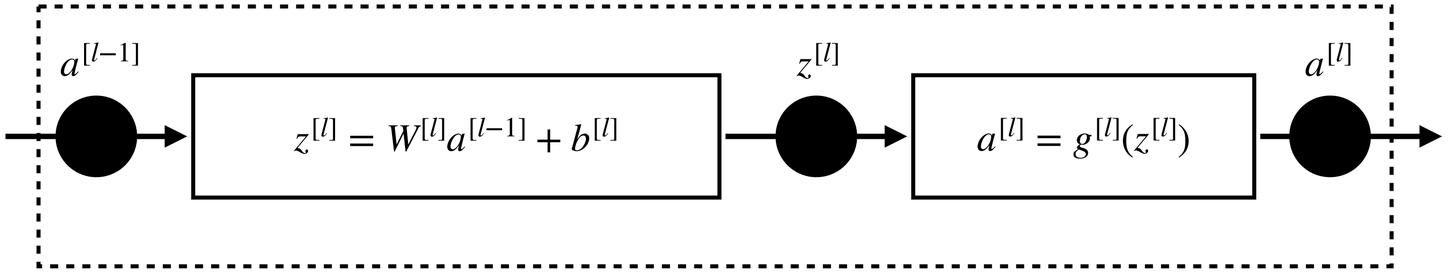


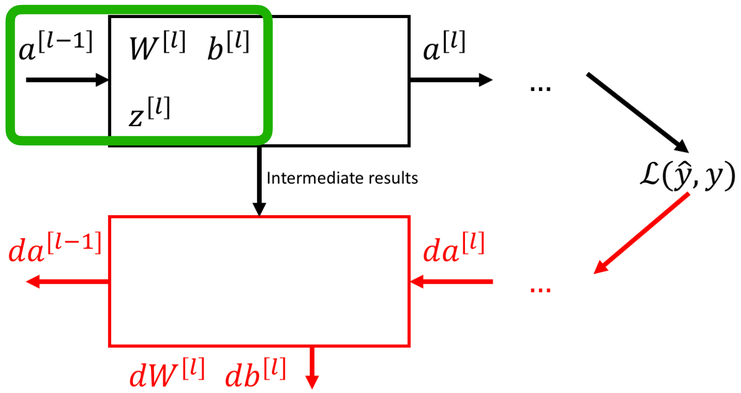
Back-propagation with Neural Network



Back-propagation with Neural Network

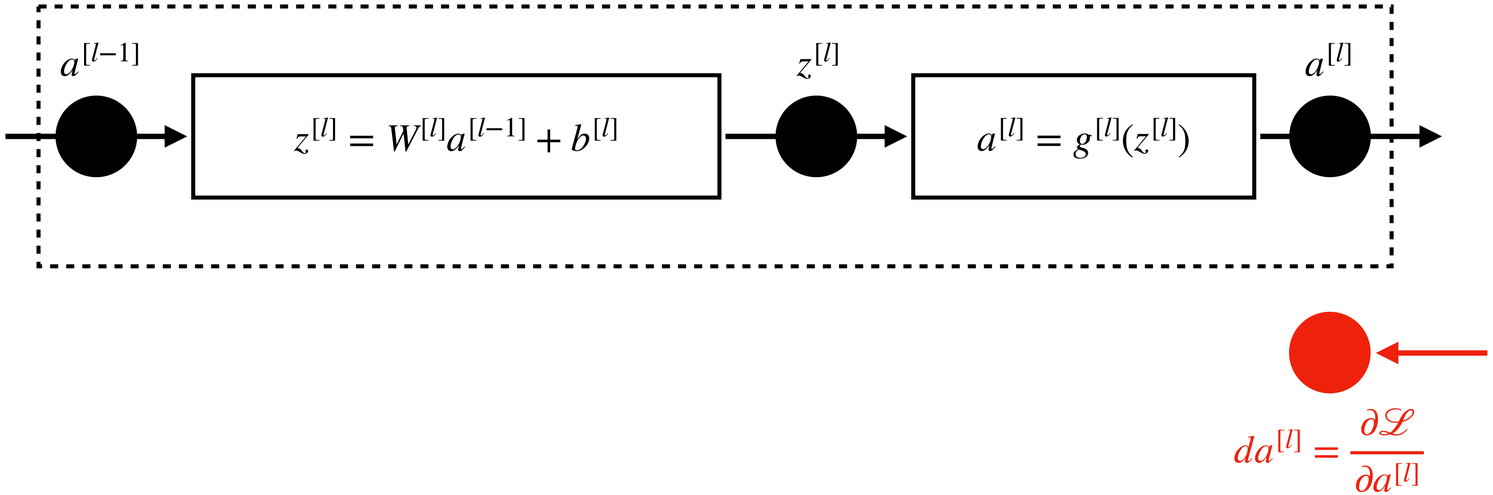
Backward Pass

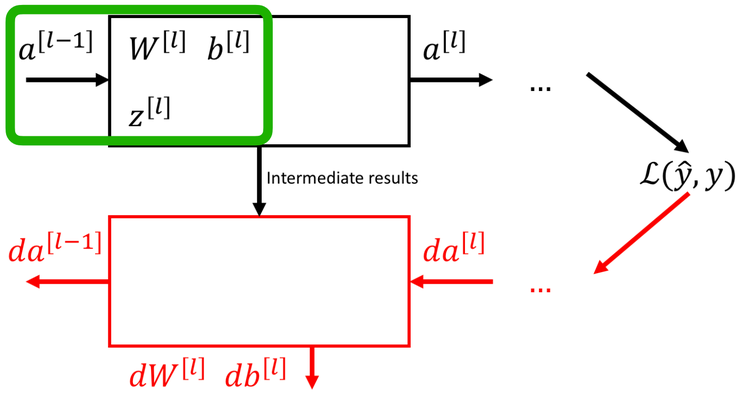




Back-propagation with Neural Network

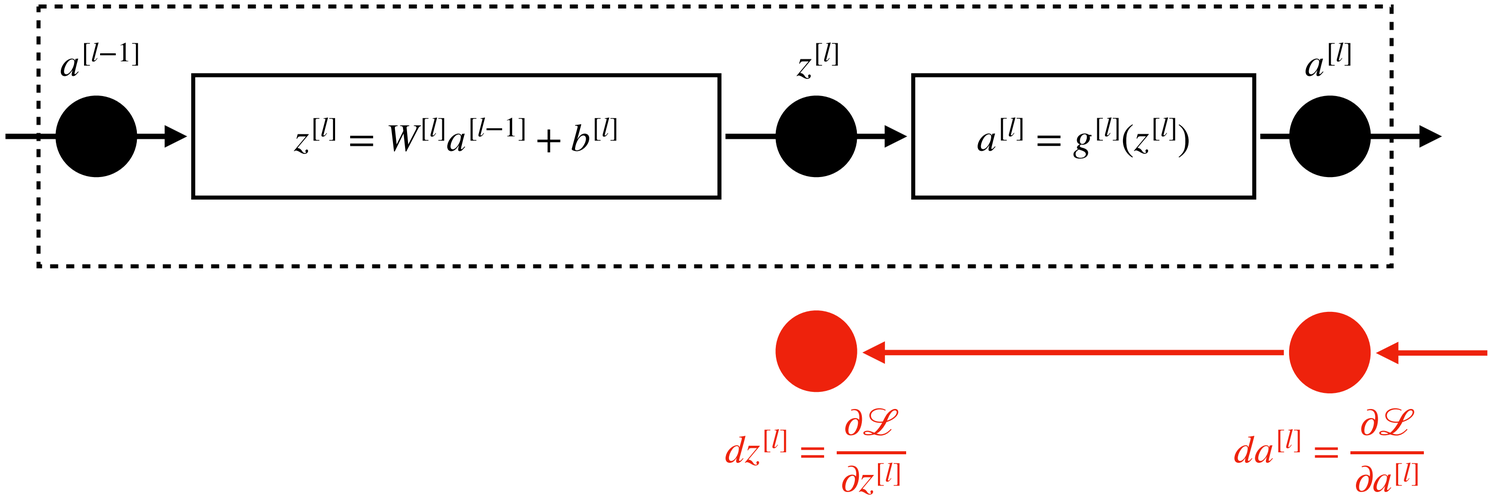
Backward Pass

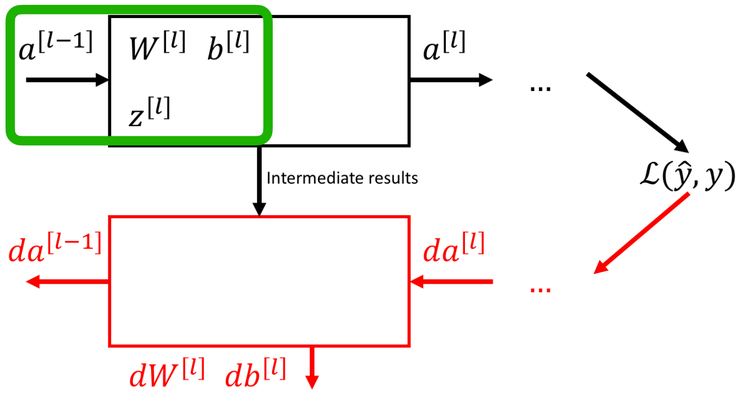




Back-propagation with Neural Network

Backward Pass





$$dz^{[l]} = g^{[l]\prime}(z^{[l]}) * da^{[l]}$$

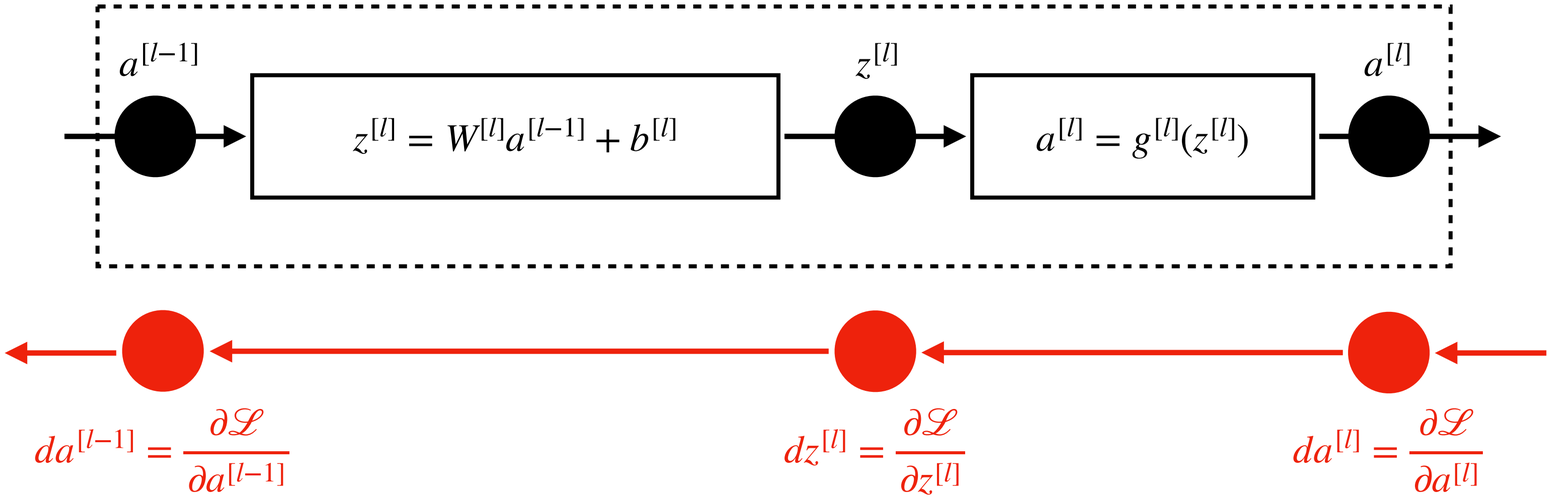
Inner function

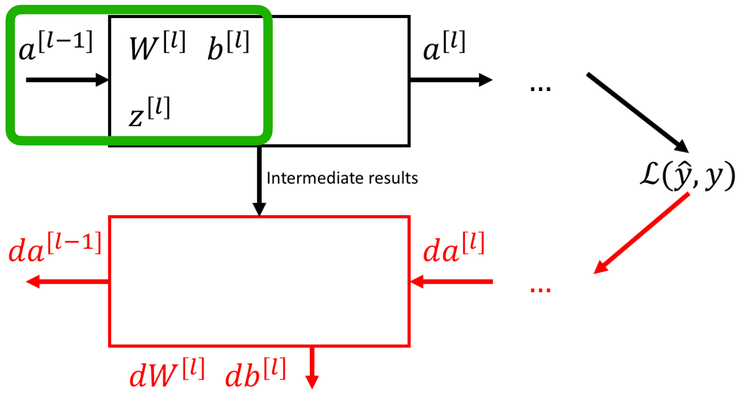
Element-wise multiplication

Outer function

Back-propagation with Neural Network

Backward Pass





$$dz^{[l]} = g^{[l]\prime}(z^{[l]}) * da^{[l]}$$

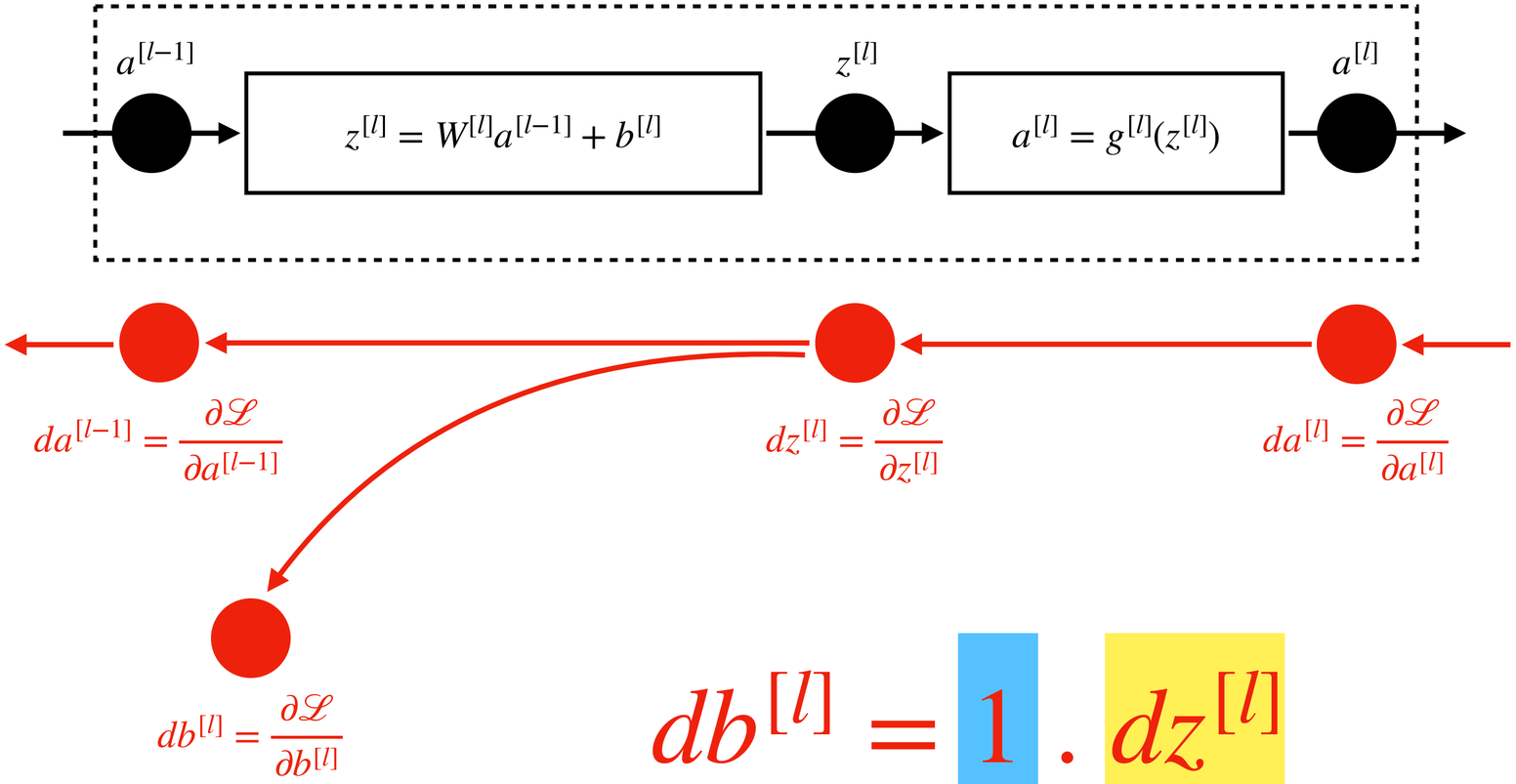
$$da^{[l-1]} = W^{[l]^T} dz^{[l]}$$

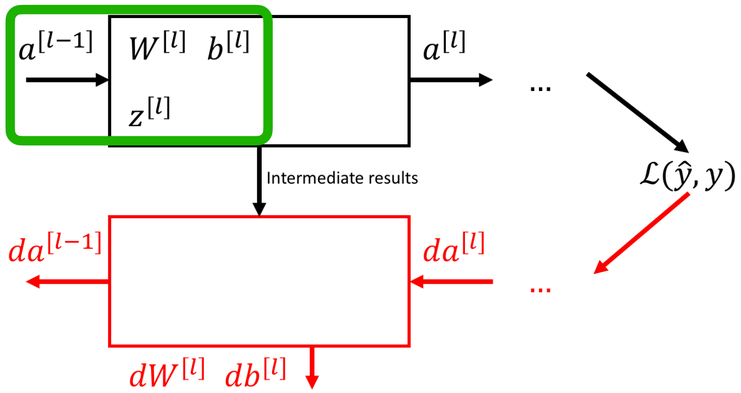
Inner function

Outer function

Back-propagation with Neural Network

Backward Pass





$$dz^{[l]} = g^{[l]\prime}(z^{[l]}) * da^{[l]}$$

$$da^{[l-1]} = W^{[l]^T} dz^{[l]}$$

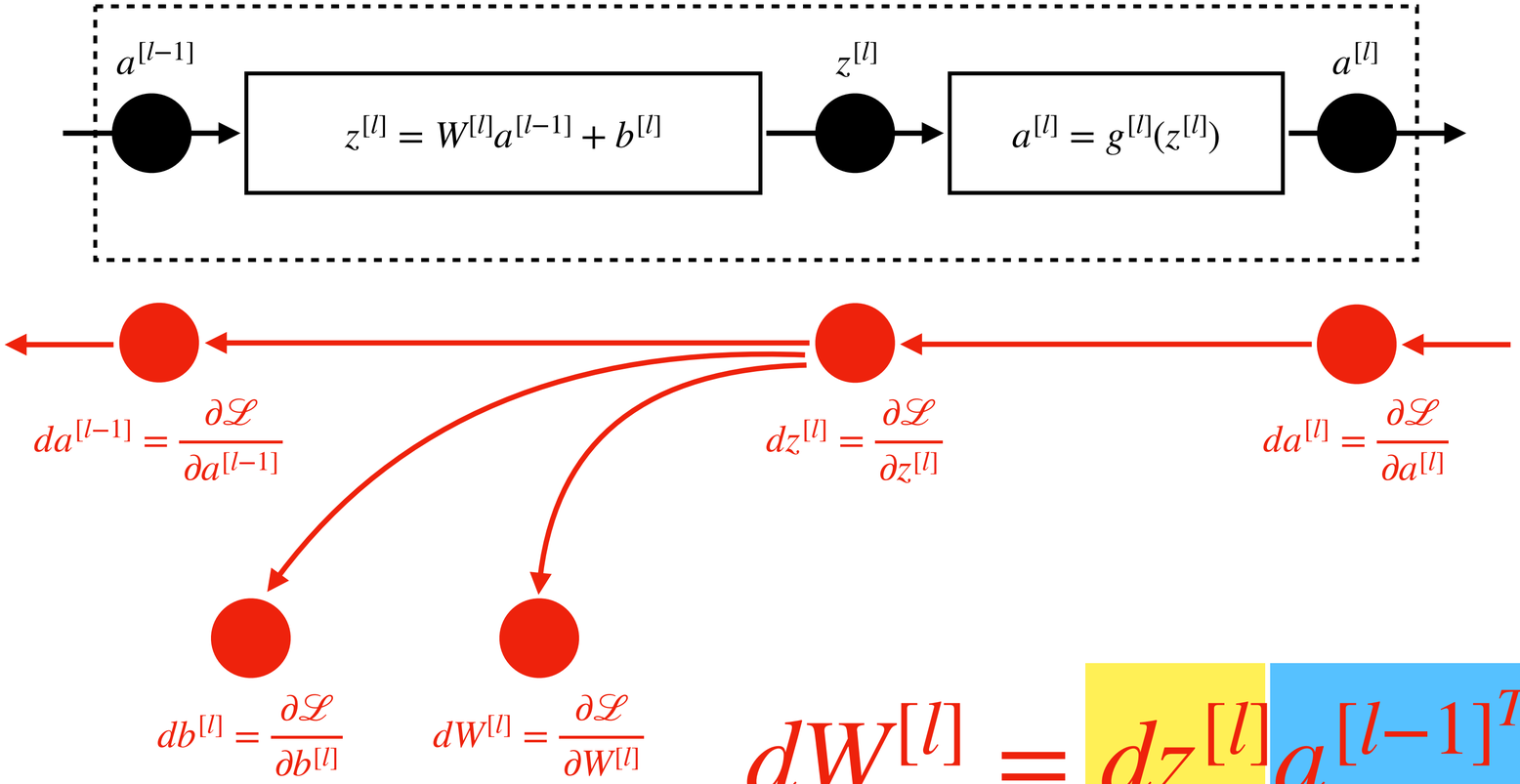
$$db^{[l]} = 1 \cdot dz^{[l]}$$

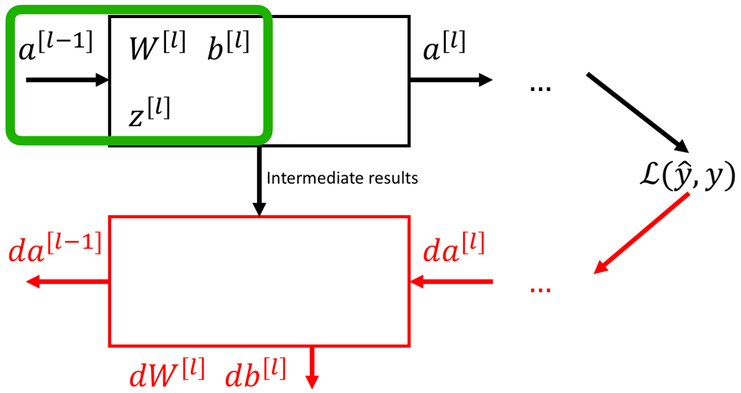
Inner function

Outer function

Back-propagation with Neural Network

Backward Pass





$$dz^{[l]} = g^{[l]\prime}(z^{[l]}) * da^{[l]}$$

$$da^{[l-1]} = W^{[l]^T} dz^{[l]}$$

$$db^{[l]} = dz^{[l]}$$

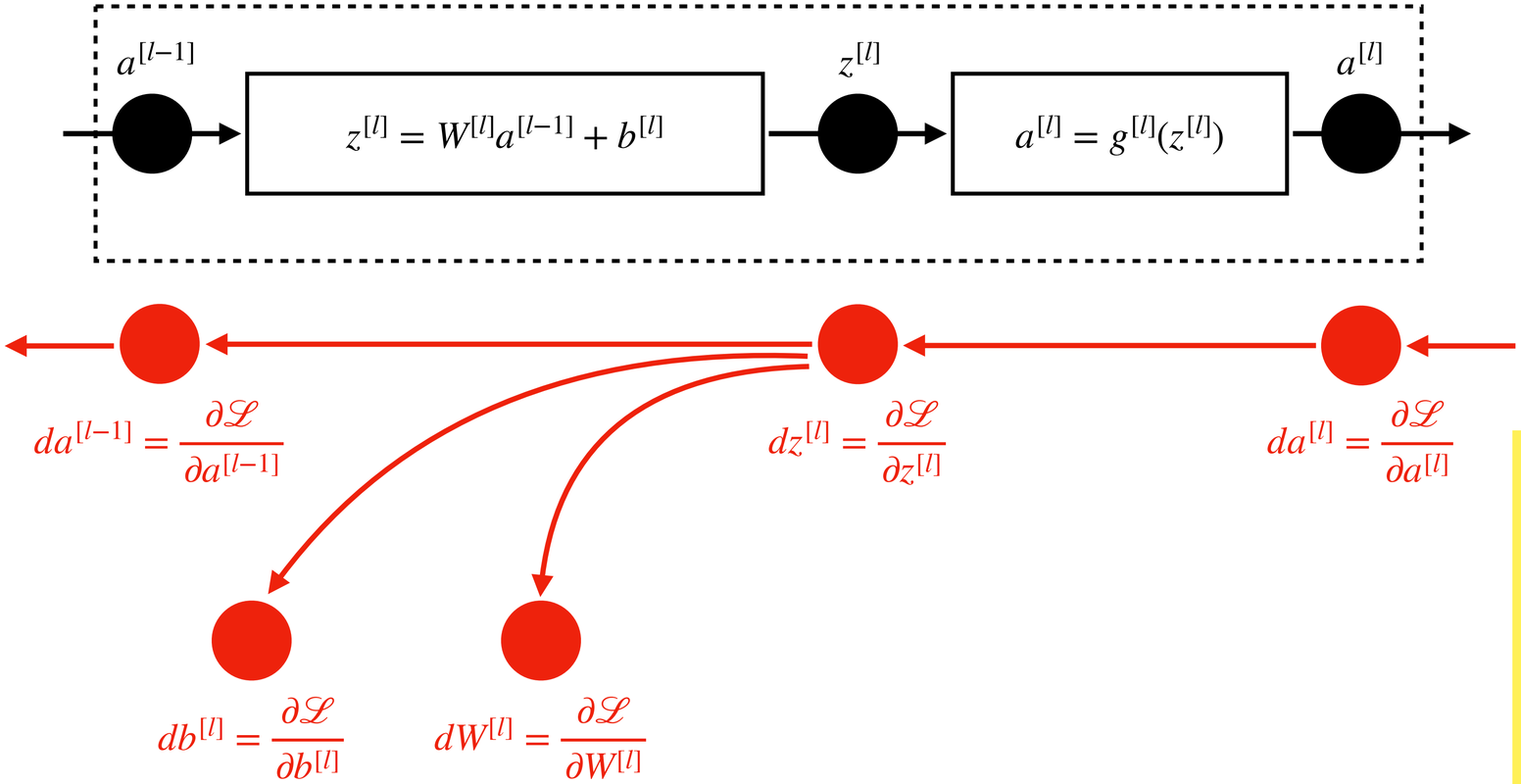
$$dW^{[l]} = dz^{[l]} a^{[l-1]^T}$$

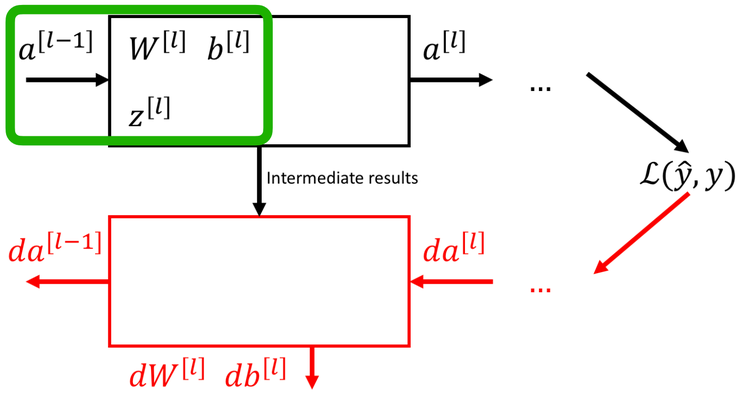
Outer function

Inner function

Back-propagation with Neural Network

Backward Pass





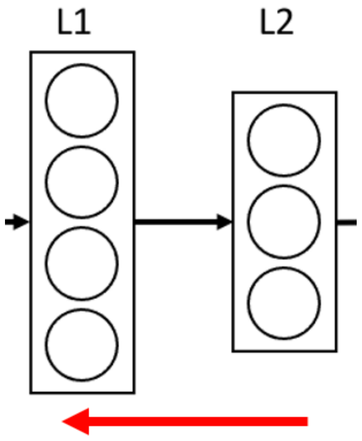
$$dz^{[l]} = g^{[l]\prime}(z^{[l]}) * da^{[l]}$$
$$da^{[l-1]} = W^{[l]^T} dz^{[l]}$$
$$db^{[l]} = dz^{[l]}$$
$$dW^{[l]} = dz^{[l]} a^{[l-1]^T}$$

Back-propagation with Neural Network

Shapes

$$dz^{[2]} = g^{[2]'}(z^{[2]}) * da^{[2]}$$

(3×1)

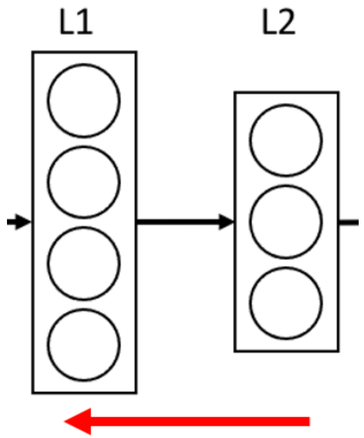


$$dz^{[l]} = g^{[l]'}(z^{[l]}) * da^{[l]}$$
$$da^{[l-1]} = W^{[l]^T} dz^{[l]}$$
$$db^{[l]} = dz^{[l]}$$
$$dW^{[l]} = dz^{[l]} a^{[l-1]^T}$$

Back-propagation with Neural Network

Shapes

$$dz^{[2]} = g^{[2]'}(z^{[2]}) * da^{[2]}$$
$$da^{[1]} = W^{[2]^T} dz^{[2]}$$



$$\begin{bmatrix} | \\ da^{[1]} \\ (4 \times 1) \\ | \end{bmatrix} = \begin{bmatrix} W^{[2]^T} \\ (4 \times 3) \end{bmatrix} \begin{bmatrix} | \\ dz^{[2]} \\ (3 \times 1) \\ | \end{bmatrix}$$

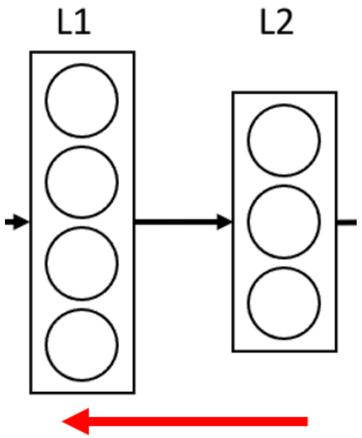
$$dz^{[l]} = g^{[l]'}(z^{[l]}) * da^{[l]}$$
$$da^{[l-1]} = W^{[l]^T} dz^{[l]}$$
$$db^{[l]} = dz^{[l]}$$
$$dW^{[l]} = dz^{[l]} a^{[l-1]^T}$$

Back-propagation with Neural Network

Shapes

$$dz^{[2]} = g^{[2]'}(z^{[2]}) * da^{[2]}$$
$$da^{[1]} = W^{[2]^T} dz^{[2]}$$
$$db^{[2]} = dz^{[2]}$$

$$\begin{bmatrix} | \\ db^{[2]} \\ (3 \times 1) \\ | \end{bmatrix} = \begin{bmatrix} | \\ dz^{[2]} \\ (3 \times 1) \\ | \end{bmatrix}$$

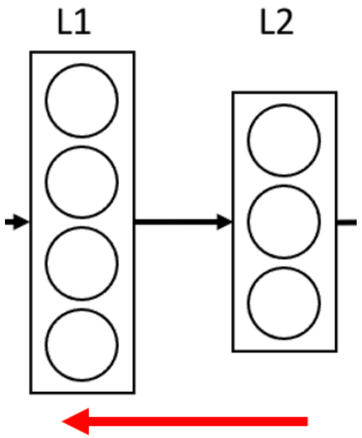


$$dz^{[l]} = g^{[l]'}(z^{[l]}) * da^{[l]}$$
$$da^{[l-1]} = W^{[l]^T} dz^{[l]}$$
$$db^{[l]} = dz^{[l]}$$
$$dW^{[l]} = dz^{[l]} a^{[l-1]^T}$$

Back-propagation with Neural Network

Shapes

$$dz^{[2]} = g^{[2]'}(z^{[2]}) * da^{[2]}$$
$$da^{[1]} = W^{[2]^T} dz^{[2]}$$
$$db^{[2]} = dz^{[2]}$$
$$dW^{[2]} = dz^{[2]} a^{[1]^T}$$



$$\begin{bmatrix} dW^{[2]} \\ (3 \times 4) \end{bmatrix} = \begin{bmatrix} | \\ dz^{[2]} \\ (3 \times 1) \\ | \end{bmatrix} \begin{bmatrix} - & a^{[1]^T} & - \\ & (1 \times 4) & \end{bmatrix}$$

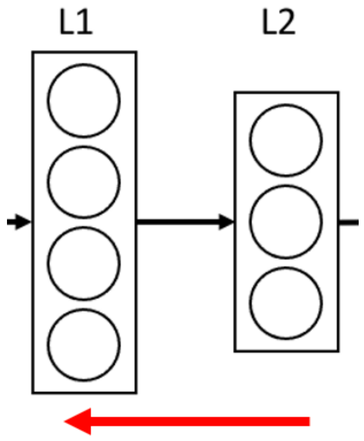
$$dz^{[l]} = g^{[l]'}(z^{[l]}) * da^{[l]}$$
$$da^{[l-1]} = W^{[l]^T} dz^{[l]}$$
$$db^{[l]} = dz^{[l]}$$
$$dW^{[l]} = dz^{[l]} a^{[l-1]^T}$$

Back-propagation with Neural Network

Shapes - Vectorized

$$dZ^{[2]} = g^{[2]\prime}(Z^{[2]}) * dA^{[2]}$$

$(3 \times m)$



$$dZ^{[l]} = g^{[l]\prime}(Z^{[l]}) * dA^{[l]}$$

$$dA^{[l-1]} = W^{[l]^T} dZ^{[l]}$$

$$db^{[l]} = mean(dZ^{[l]})$$

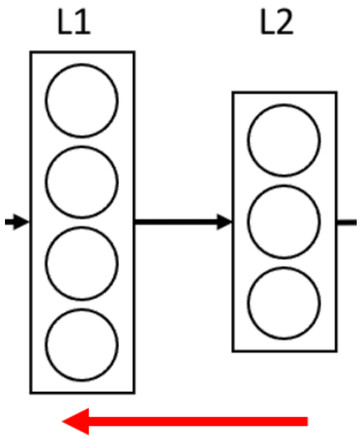
$$dW^{[l]} = mean(dZ^{[l]} A^{[l-1]^T})$$

Back-propagation with Neural Network

Shapes - Vectorized

$$dZ^{[2]} = g^{[2]'}(Z^{[2]}) * dA^{[2]}$$

$$dA^{[1]} = W^{[2]^T} dZ^{[2]}$$



$$\begin{bmatrix} \cdots & | & \cdots \\ \cdots & dA^{[1]} & \cdots \\ & (4 \times m) & \\ & | & \end{bmatrix} = \begin{bmatrix} W^{[2]^T} \\ (4 \times 3) \end{bmatrix} \begin{bmatrix} \cdots & | & \cdots \\ \cdots & dZ^{[2]} & \cdots \\ & (3 \times m) & \\ & | & \end{bmatrix}$$

$$dZ^{[l]} = g^{[l]'}(Z^{[l]}) * dA^{[l]}$$

$$dA^{[l-1]} = W^{[l]^T} dZ^{[l]}$$

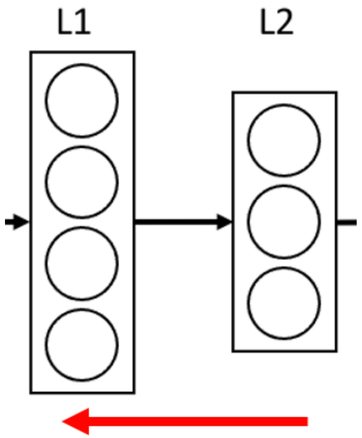
$$db^{[l]} = mean(dZ^{[l]})$$

$$dW^{[l]} = mean(dZ^{[l]} A^{[l-1]^T})$$

Back-propagation with Neural Network

Shapes - Vectorized

$$dZ^{[2]} = g^{[2]\prime}(Z^{[2]}) * dA^{[2]}$$
$$dA^{[1]} = W^{[2]^T} dZ^{[2]}$$
$$db^{[2]} = mean(dZ^{[2]})$$



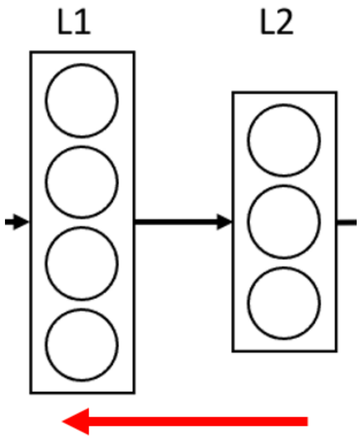
$$\begin{bmatrix} | \\ db^{[2]} \\ (3 \times 1) \\ | \end{bmatrix} = \frac{1}{m} \overbrace{\begin{bmatrix} \dots & | & \dots \\ \dots & dZ^{[2]} & \dots \\ & (3 \times m) & \\ & | & \end{bmatrix}}^{m} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ (m \times 1) \end{bmatrix} \Bigg\} m$$

$$dZ^{[l]} = g^{[l]\prime}(Z^{[l]}) * dA^{[l]}$$
$$dA^{[l-1]} = W^{[l]^T} dZ^{[l]}$$
$$db^{[l]} = mean(dZ^{[l]})$$
$$dW^{[l]} = mean(dZ^{[l]} A^{[l-1]^T})$$

Back-propagation with Neural Network

Shapes - Vectorized

$$dZ^{[2]} = g^{[2]\prime}(Z^{[2]}) * dA^{[2]}$$
$$dA^{[1]} = W^{[2]^T} dZ^{[2]}$$
$$db^{[2]} = mean(dZ^{[2]})$$
$$dW^{[2]} = mean(dZ^{[2]} A^{[1]^T})$$

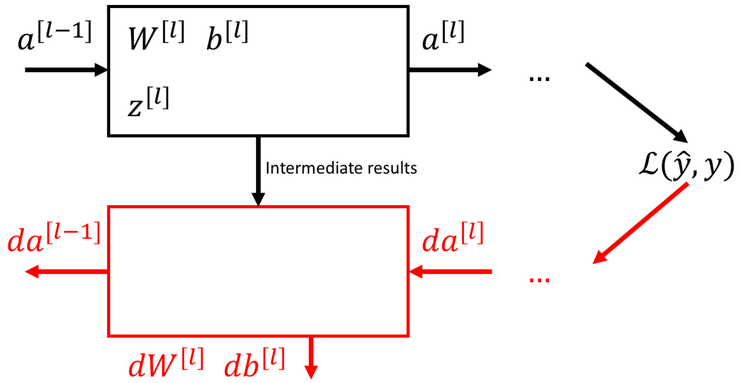


$$\begin{bmatrix} dW^{[2]} \\ (3 \times 4) \end{bmatrix} = \frac{1}{m} \overbrace{\begin{bmatrix} \cdots & | & \cdots \\ \cdots & dZ^{[2]} & \cdots \\ & (3 \times m) & \\ & | & \end{bmatrix}}^{m} \begin{bmatrix} - & A^{[1]^T} & - \\ \cdots & (m \times 4) & \cdots \\ \cdots & & \cdots \\ \cdots & & \cdots \end{bmatrix} \Bigg\} m$$

$$dZ^{[l]} = g^{[l]\prime}(Z^{[l]}) * dA^{[l]}$$
$$dA^{[l-1]} = W^{[l]^T} dZ^{[l]}$$
$$db^{[l]} = mean(dZ^{[l]})$$
$$dW^{[l]} = mean(dZ^{[l]} A^{[l-1]^T})$$

Back-propagation with Neural Network

Shapes - Vectorized




```python
def backward(self, da, a_prev, intermediate_results):

    W, b, z = intermediate_results     # From forward pass
    _, m = da.shape                    # Get number of samples - m

    # Find activation gradient for previous layer
    dz = np.multiply(self.activation.derivative(z), da)
    da_prev = np.matmul(W.T, dz)

    # Find gradients for weights and biases
    db = (1.0/m) * np.sum(dz, axis=1, keepdims=True)
    dW = (1.0/m) * np.matmul(dz, a_prev.T)
```

$$dZ^{[l]} = g^{[l]\prime}(Z^{[l]}) * dA^{[l]}$$
$$dA^{[l-1]} = W^{[l]^T} dZ^{[l]}$$
$$db^{[l]} = mean(dZ^{[l]})$$
$$dW^{[l]} = mean(dZ^{[l]} A^{[l-1]^T})$$

2 Machine Learning Project

Machine Learning Project

Goal

- **What** is it that we need to accomplish ?

Machine Learning Project

Goal

- **What** is it that we need to accomplish ?

Identify breed of cats

Machine Learning Project

Goal

- **What** is it that we need to accomplish ?

Identify breed of cats

Classification

Machine Learning Project

Goal

- **What** is it that we need to accomplish ?

Identify breed of cats

Classification

Binary ?
Multi-class ?

Machine Learning Project

Metrics

- **How can we tell** if we are doing **better or worse** ?
- Metrics and Loss functions are not the same !

Machine Learning Project

Metrics

- **How can we tell** if we are doing **better or worse** ?
- Metrics and Loss functions are not the same !
    - We need a single quantity that can be put on a scale

Machine Learning Project

Metrics

- **How can we tell** if we are doing **better or worse** ?
- Metrics and Loss functions are not the same !
    - We need a single quantity that can be put on a scale

	Real True	Real False
Predicted True	TP	FP
Predicted False	FN	TN

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions}$$

Machine Learning Project

Metrics

- **How can we tell** if we are doing **better or worse** ?
- Metrics and Loss functions are not the same !
    - We need a single quantity that can be put on a scale

	Real True	Real False
Predicted True	TP	FP
Predicted False	FN	TN

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Machine Learning Project

Metrics

- **How can we tell** if we are doing **better or worse** ?
- Metrics and Loss functions are not the same !
    - If multiple aspects are important, we need some kind of **compound score**

	Real True	Real False
Predicted True	TP	FP
Predicted False	FN	TN

$$Precision = \frac{TP}{TP + FP}$$

Machine Learning Project

Metrics

- **How can we tell** if we are doing **better or worse** ?
- Metrics and Loss functions are not the same !
    - If multiple aspects are important, we need some kind of **compound score**

	Real True	Real False
Predicted True	TP	FP
Predicted False	FN	TN

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

Machine Learning Project

Metrics

- **How can we tell** if we are doing **better or worse** ?
- Metrics and Loss functions are not the same !
    - If multiple aspects are important, we need some kind of **compound score**

	Real True	Real False
Predicted True	TP	FP
Predicted False	FN	TN

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1\ Score = \frac{TP}{TP + \frac{1}{2}(FP + FN)}$$

Machine Learning Project

Data

- **Validation Set** is of paramount importance !
- Represents the **real use-case**
- Can be difficult to obtain

Machine Learning Project

Data

- **Validation Set** is of paramount importance !
- Represents the **real use-case**
- Can be **difficult to obtain**

Make sure your validation data represents your intended use case

Always make sure your validation data represents your intended use case

Machine Learning Project

Data

- **Train Set** should be as close to the validation set as possible
- **Can** be different !
    - Downloaded from Internet
    - Different task
    - Synthetic
    - …
- Cheap / Cheaper to obtain

Machine Learning Project

Data

- **Test Set** is typically not public
- Contains real data of your customers
    - Used for evaluation upon delivery

Machine Learning Project

Data

Training Data

Machine Learning Project

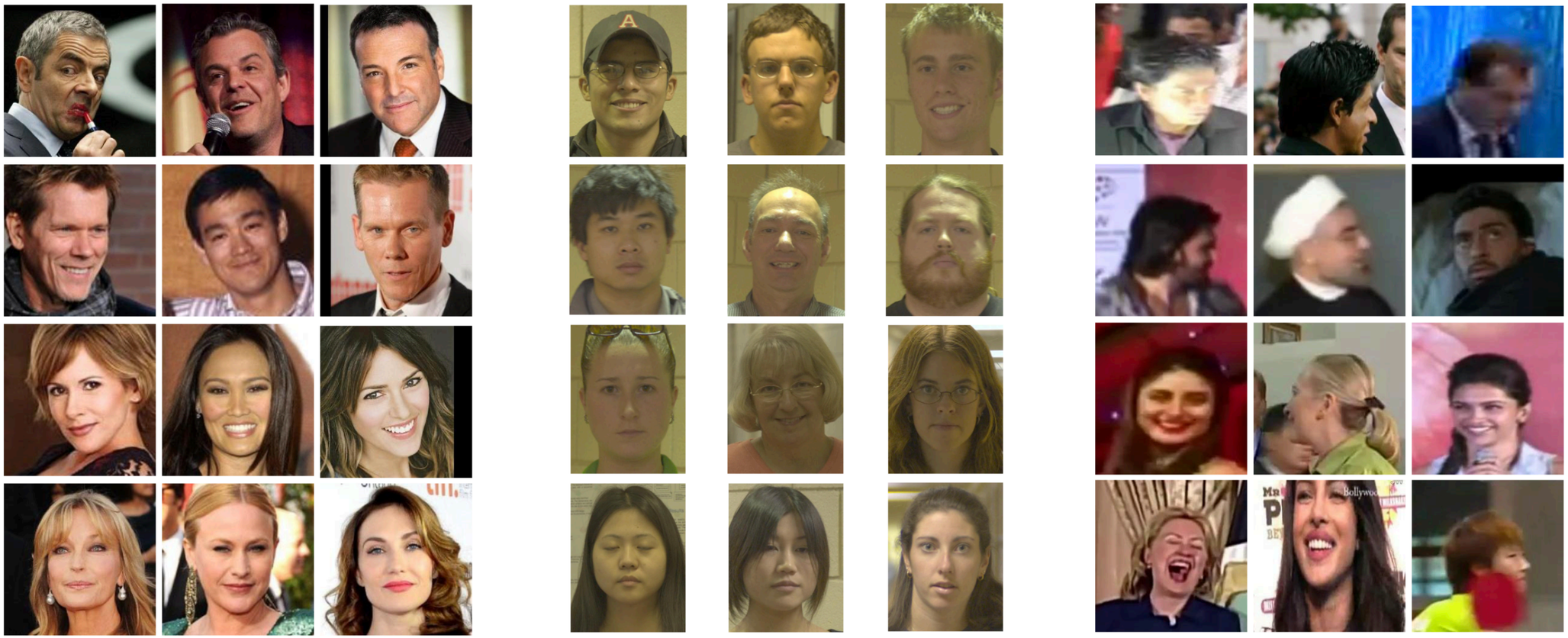
Data

Training Data

Customer Data

Machine Learning Project

Data - Domain Adaptation



(a) CASIA-Webface (b) GBU (c) IJB-A

Wang, Mei, and Weihong Deng. "Deep face recognition with clustering based domain adaptation." *Neurocomputing* 393 (2020): 1-14.

Machine Learning Project

Data

- There is **no single best solution**
- Each and every use-case is different
- Trained model will perform better when production data is close to validation data
- During the training, validation set prevents overfitting on train set

Machine Learning Project

Model Architecture

- Any model with **sufficient capacity** should be able to learn **on train set**
- Depending on the task and how much data you have
    - Fully connected model
    - Convolutional model
    - Transformer-based model
    - …
- Start with a small model, increase if necessary

Machine Learning Project

Training Process

- Decide which **training paradigm** is most suitable

Machine Learning Project

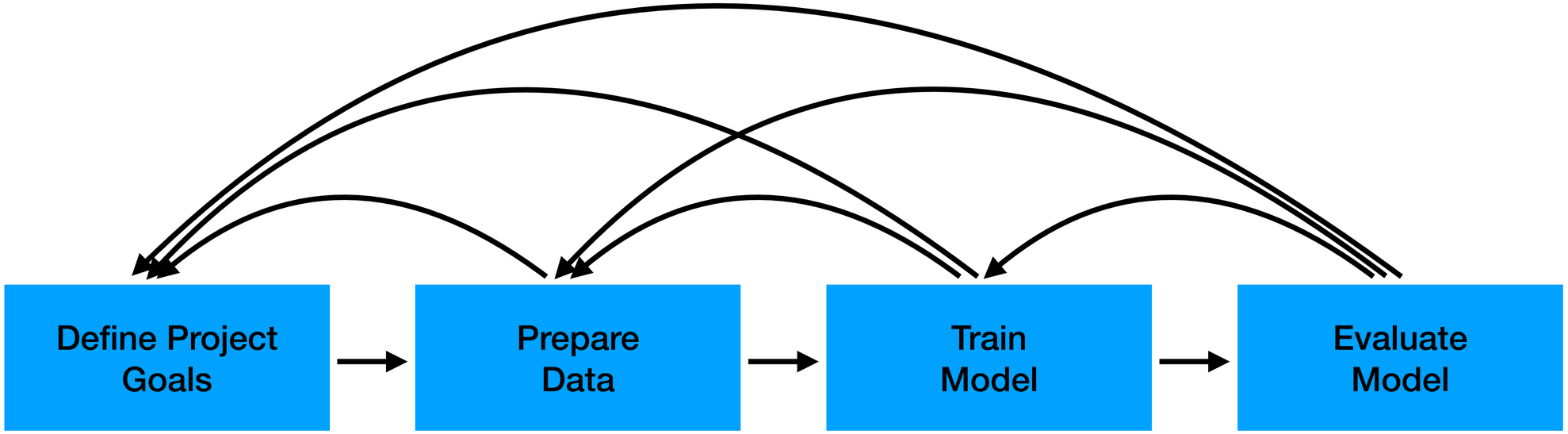
Training Process

- Decide which **training paradigm** is most suitable

    1. Supervised learning
    2. Unsupervised learning
    3. Weakly/semi supervised learning
    4. Self-supervised learning
    5. Reinforcement learning

Machine Learning Project

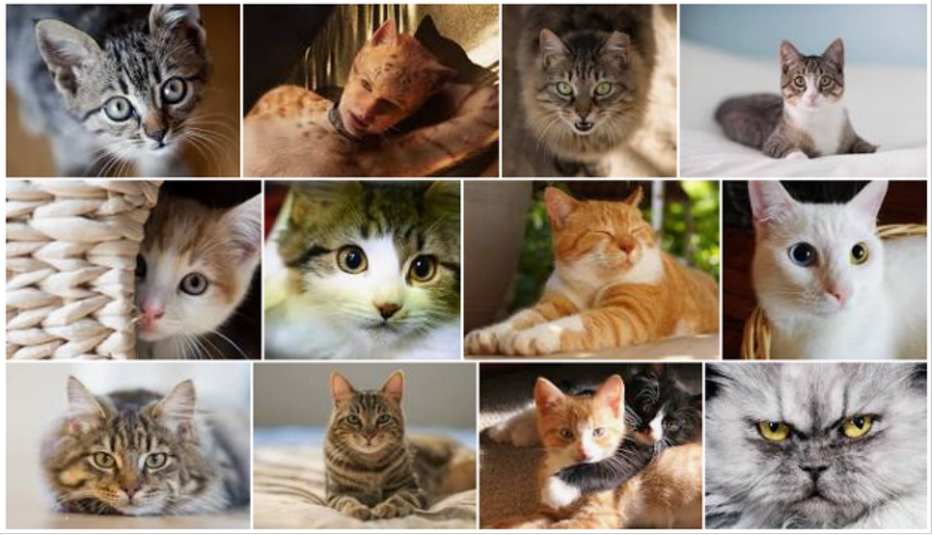
Life Cycle



Machine Learning Project

Bias / Variance

- Cat / Dog
  - Human Error ~ 0%
  - **Optimal Error**



Train Set Error	1 %	13 %	15 %	0.5 %
Validation Set Error	15 %	14 %	30 %	0.7 %
	High Variance	High Bias	High Bias High Variance	Low Bias Low Variance

Machine Learning Project

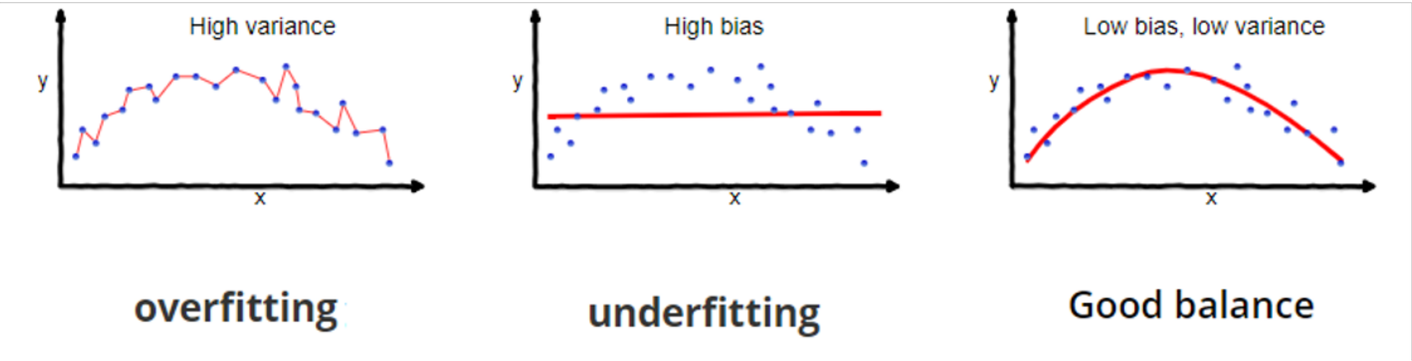
Bias / Variance

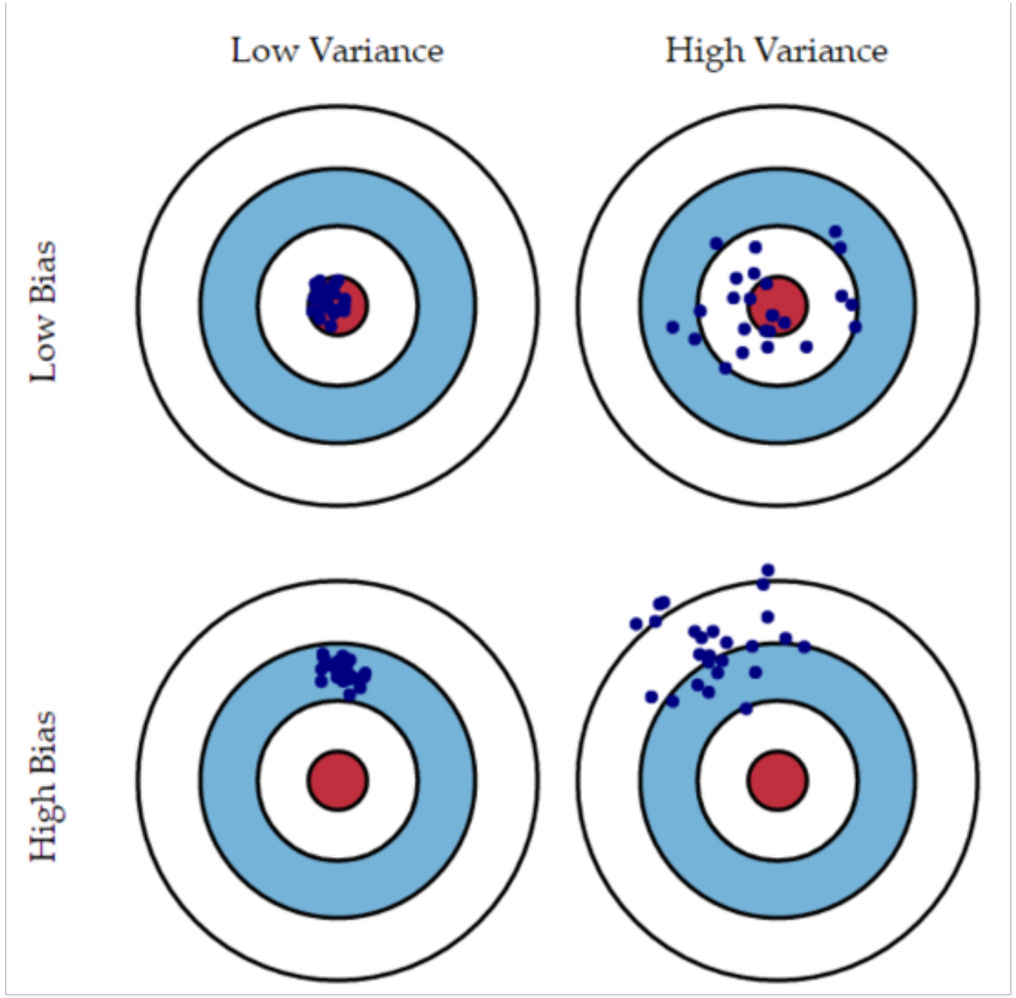
- Muffin / Chihuahua
    - **Human Error ~ 12%**
    - **Optimal Error**

Train Set Error		13 %	15 %	
Validation Set Error		14 %	30 %	
		Low Bias Low Variance	Low Bias High Variance	

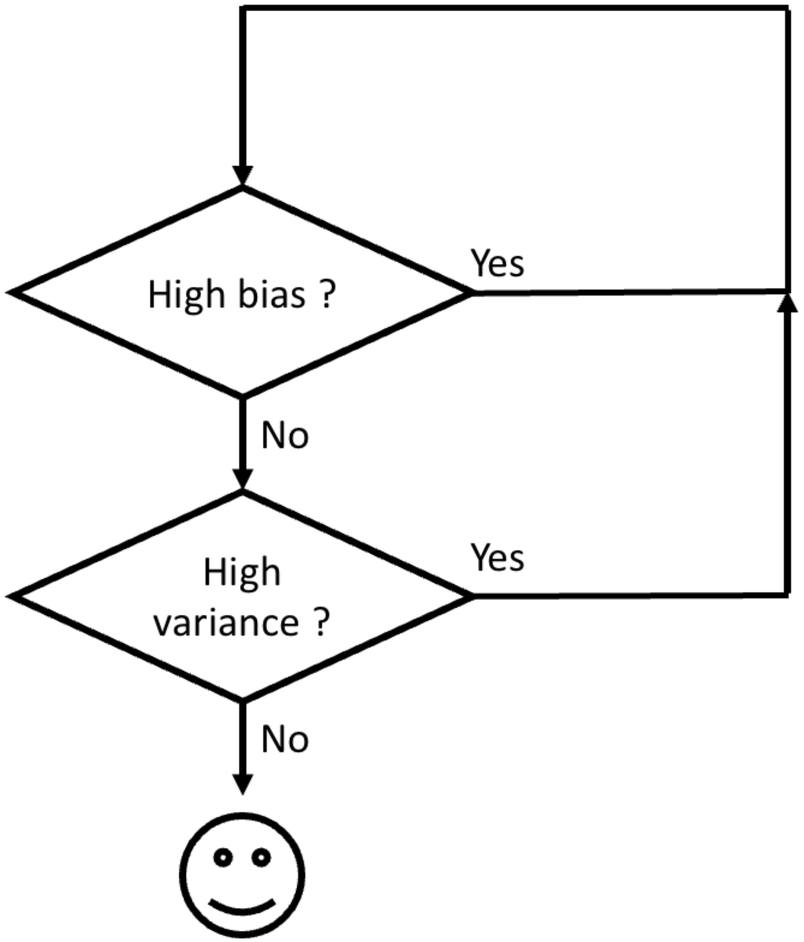
Machine Learning Project

Bias / Variance





Machine Learning Project

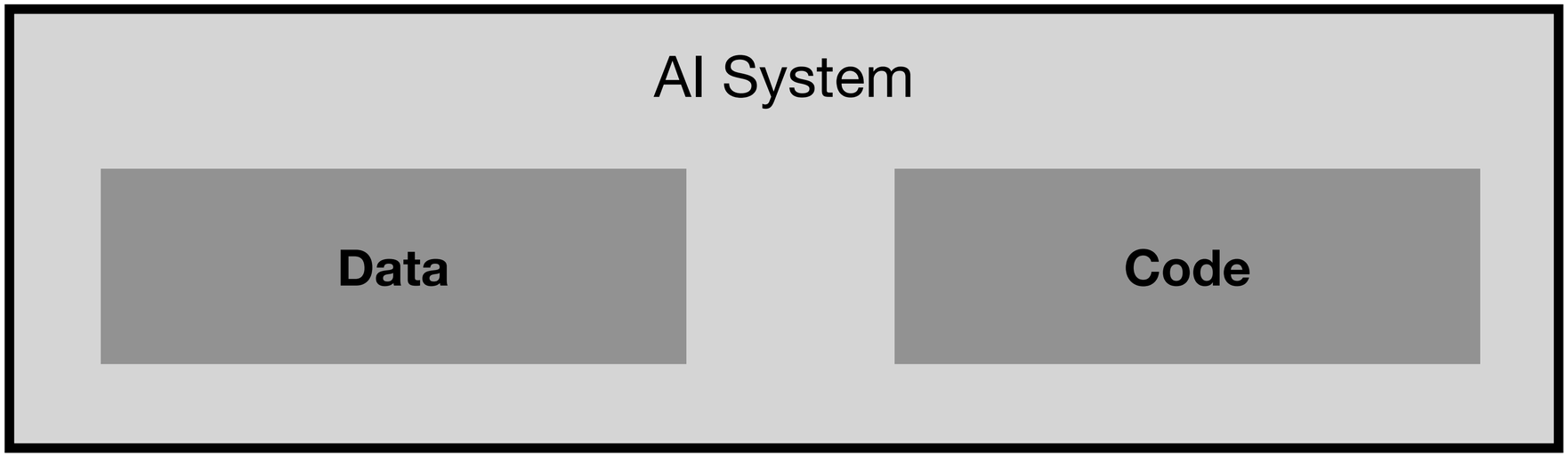


- Bigger model
- More suitable architecture
- Train longer
- Different hyper-parameters

- More data
- Regularization
- More suitable architecture

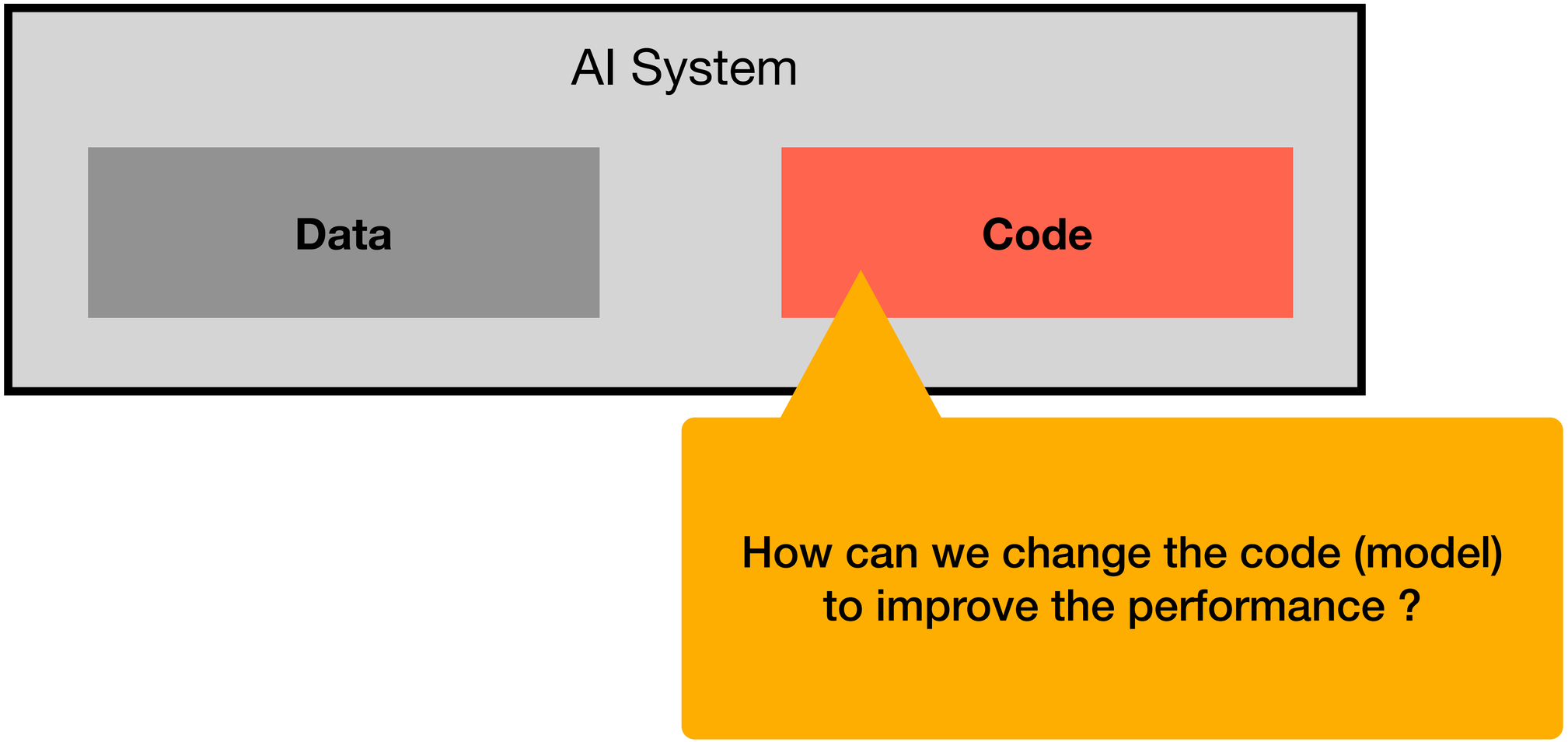
Machine Learning Project

Model Centric vs. Data Centric



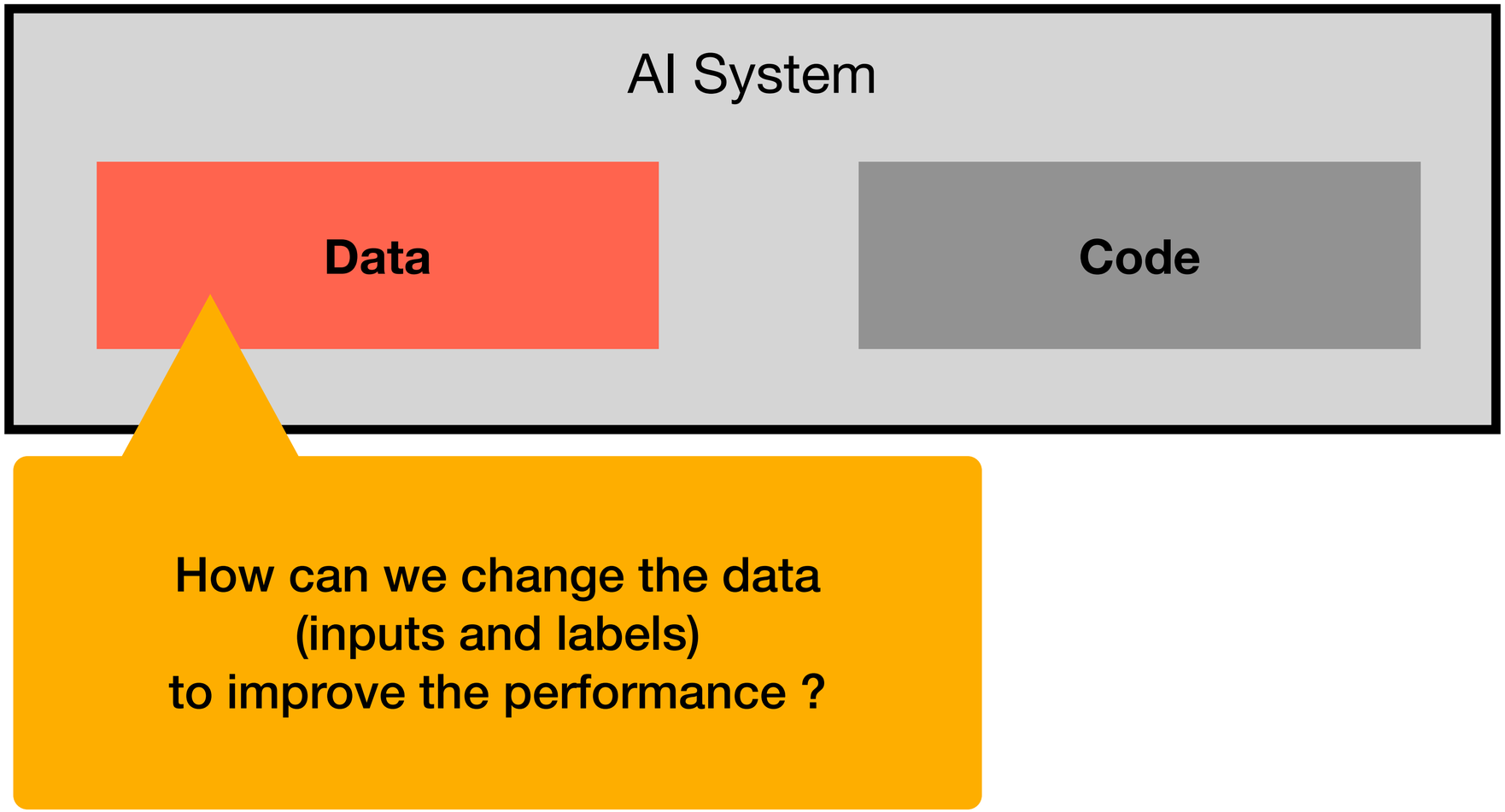
Machine Learning Project

Model Centric vs. Data Centric



Machine Learning Project

Model Centric vs. Data Centric



Machine Learning Project

Summary

- We need to have a **clear goal**
- We need proper **metrics** to tell us which solution is better
- **Validation data represents the intended use-case**
- Be aware of differences between train and validation sets
- Start with a smaller model
- Improve in iterations
- Consider improving both - model and data

3 NN Training Techniques

Training Techniques

Paradigms

1. Supervised learning
2. Unsupervised learning
3. Weakly/semi supervised learning
4. Self-supervised learning
5. Reinforcement learning

Paradigms

Supervised Learning

- For each input x_i we have ground truth annotation y_i
- Loss - measures error between predicted $\widehat{y_i} = f(x_i)$ and y_i
- Regression, Classification

Paradigms

Unsupervised Learning

- We only have inputs x_i with no annotations
- Learns patterns and structure from unlabeled data
- Self-organizing Maps
- Principal Component Analysis (PCA)
- Clustering
- Auto-Encoder

Paradigms

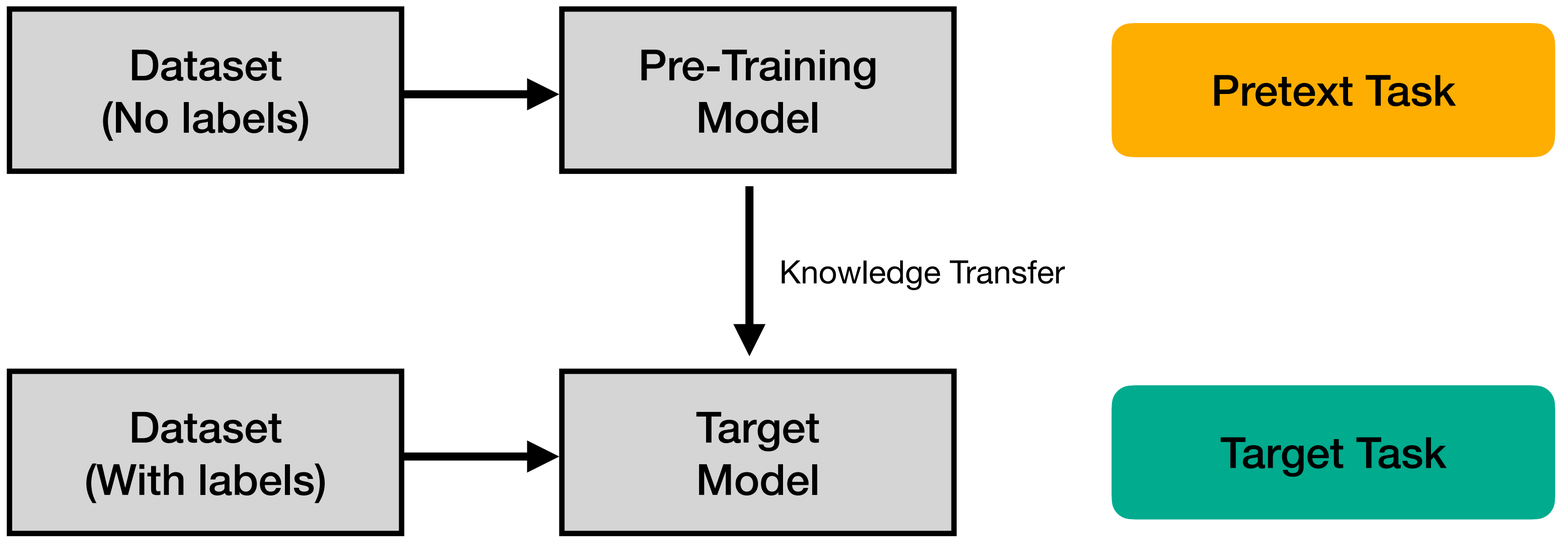
Weakly Supervised Learning

- We have **large amount** of data x_i with **no annotations**
- We have **small amount** of data x_i **with labels** y_i
- Noisy data - unreliable labels
- Training in two stages
    - Stage 1 - Supervised
    - Stage 2 - Unsupervised

Paradigms

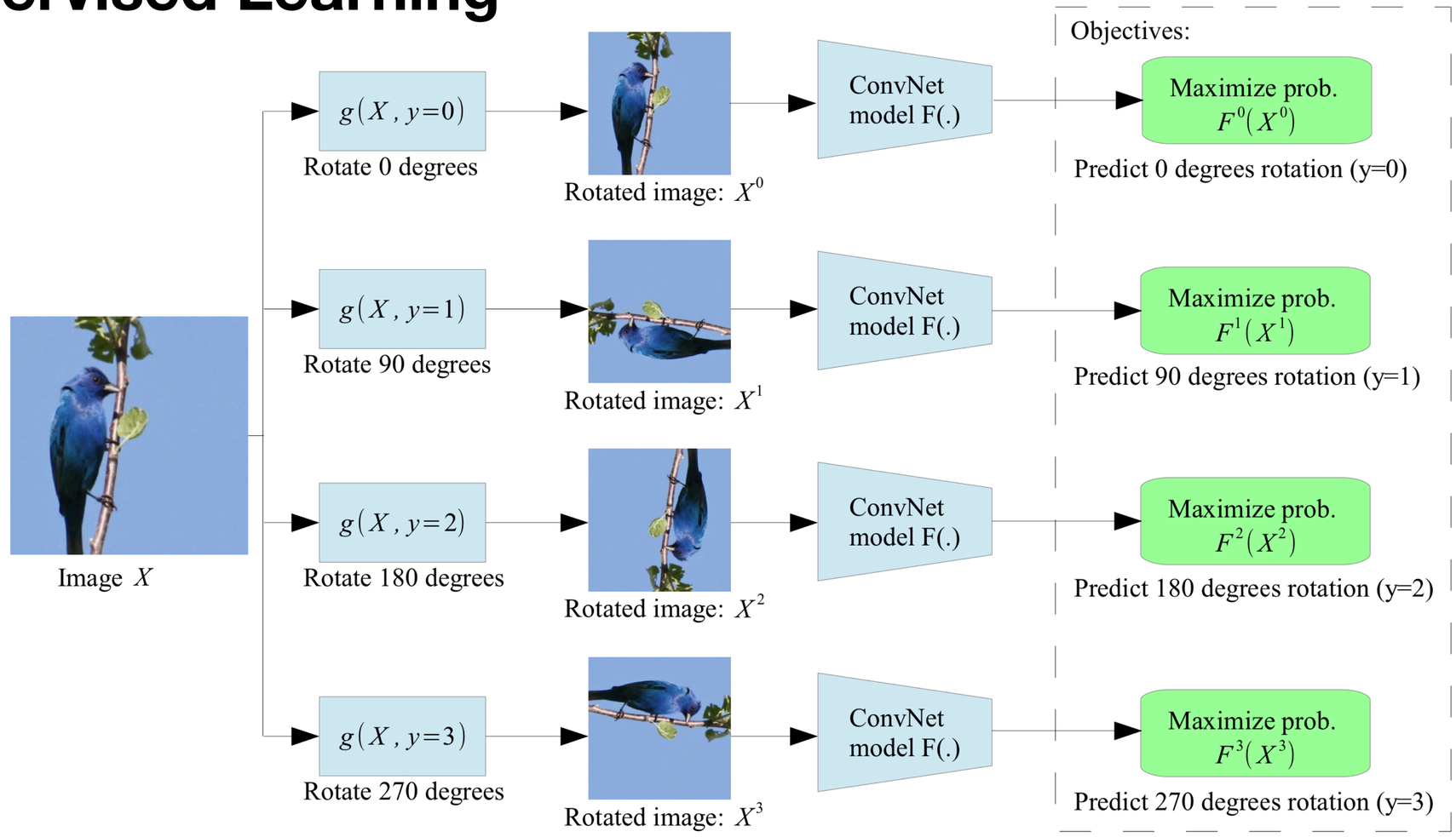
Self-Supervised Learning

- We have **large amount** of data x_i with **no annotations**
- We have **small amount** of data x_i **with labels** y_i



Paradigms

Self-Supervised Learning

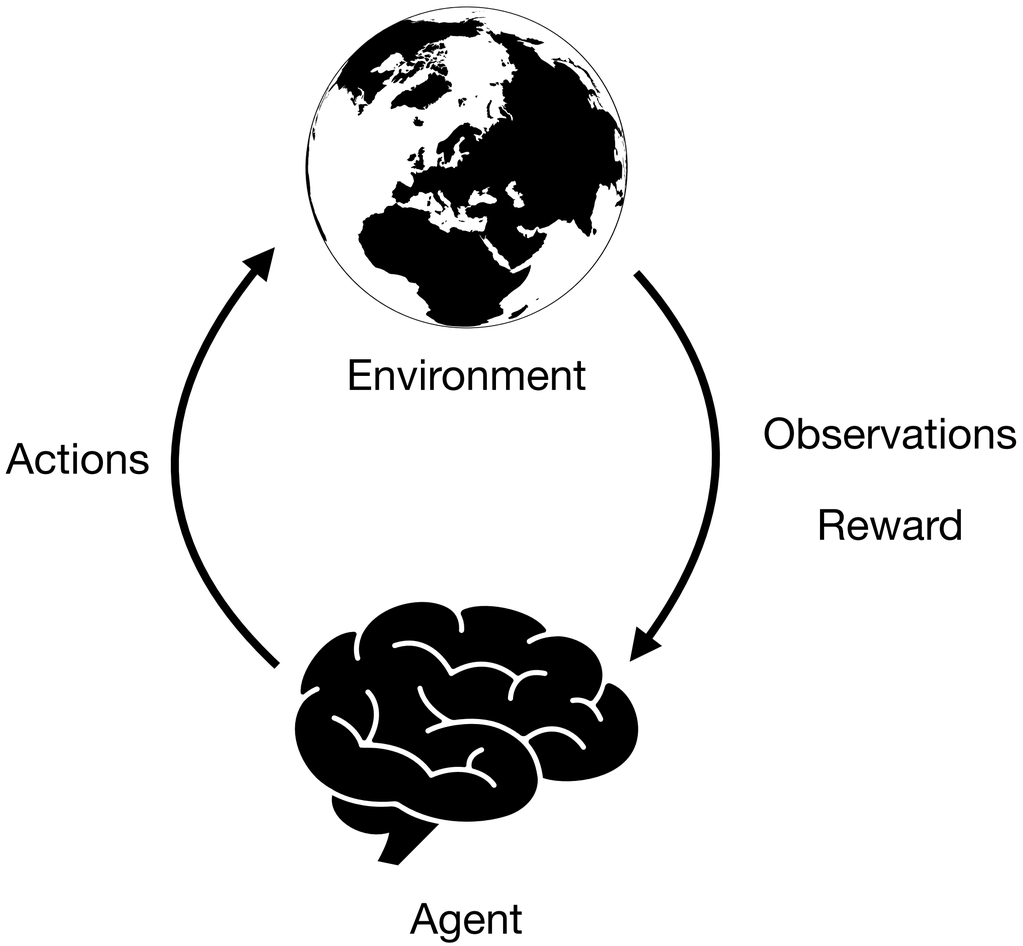


Gidaris, Spyros, Praveer Singh, and Nikos Komodakis. "Unsupervised representation learning by predicting image rotations." *arXiv preprint arXiv:1803.07728* (2018).

Paradigms

Reinforcement Learning

- Agent learns through **interaction** with the Environment
- Goal is to maximize the **total reward**



Training Techniques

Training Loop - Definition of Terms

- **Gradient descent** - algorithm, iterate in small steps until convergence
- **Dataset** - too big to fit in memory, broken down into smaller mini-batches
- **Mini-Batch** - subset of train set, contains m training samples
- **Iteration** - one step of computation
    - forward pass, backward pass, update parameters
    - computed on mini-batches (vectorized computation)
- **Epoch** - one complete pass of iterations over all mini-batches from train set

Training Techniques

Batch Size

1. Stochastic descent
    - batch size = 1
2. Mini-batch descent
    - batch size > 1
3. Batch descent
    - batch size = $M = |\mathcal{X}|$

	Batch Size	Steps / Epoch
Batch descent	M	1
Mini-batch descent	BS = 128	M / BS
Stochastic descent	1	M

Training Techniques

Batch Size

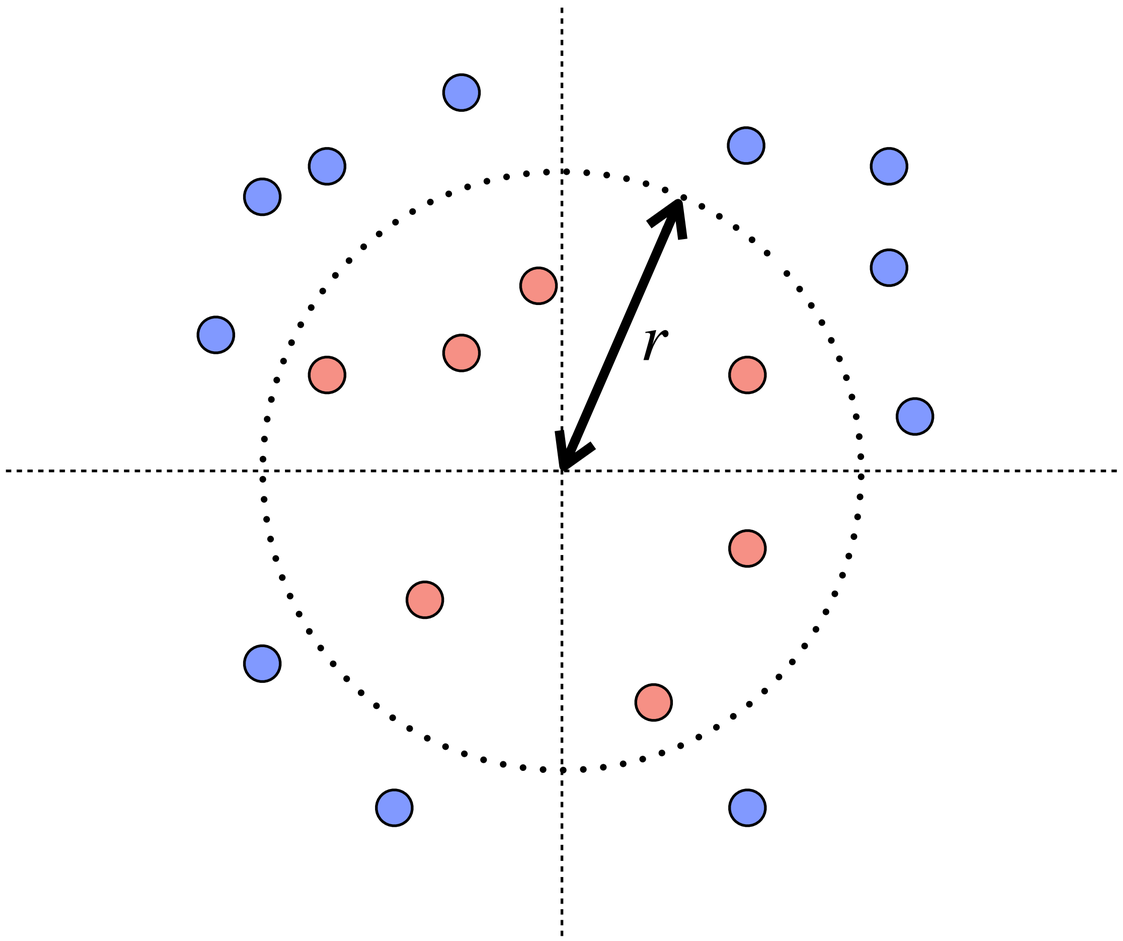
- Training - **Samples in Mini-batch must be chosen randomly**

- Validation - Order of samples does not matter

Example Toy Project

Example Toy Project

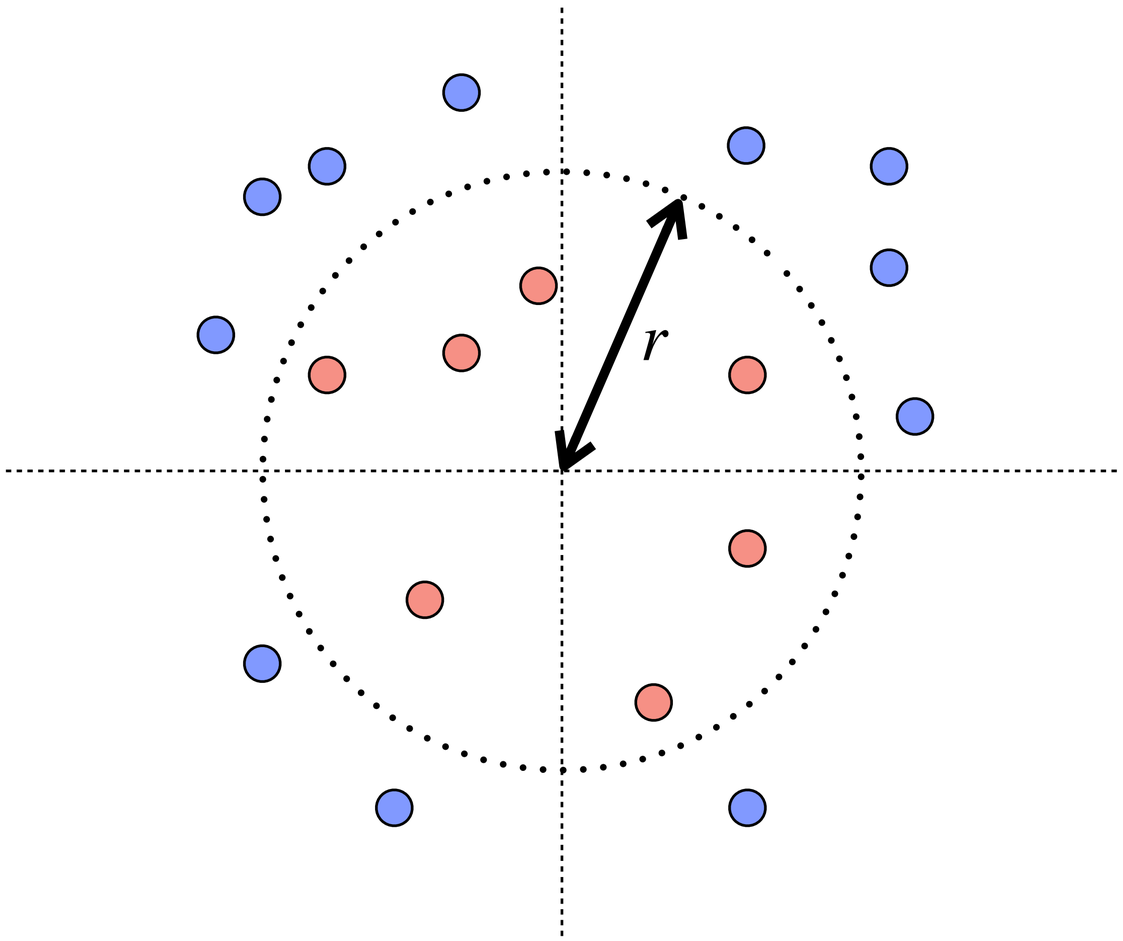
- **Goal** - What are we going to do ?



Example Toy Project

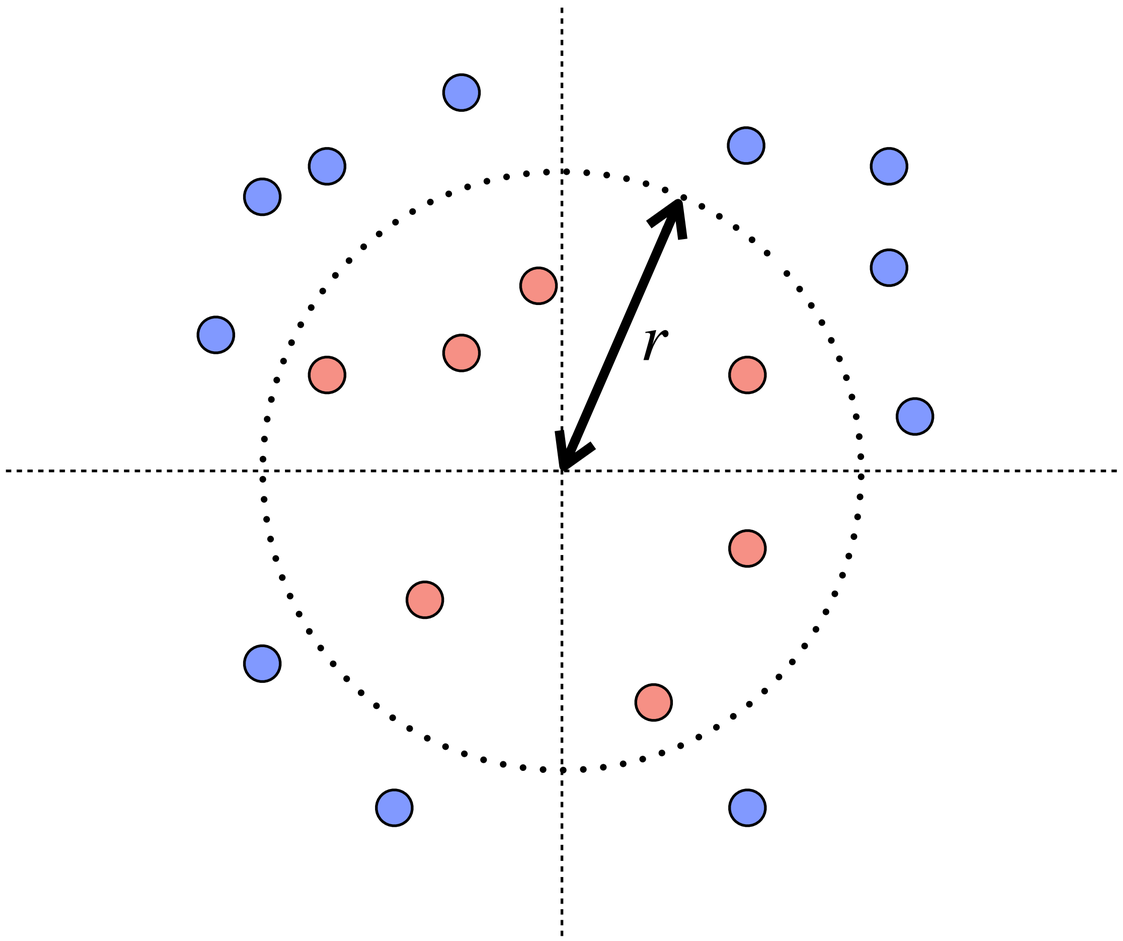
- **Goal** - What are we going to do ?

Binary Classification



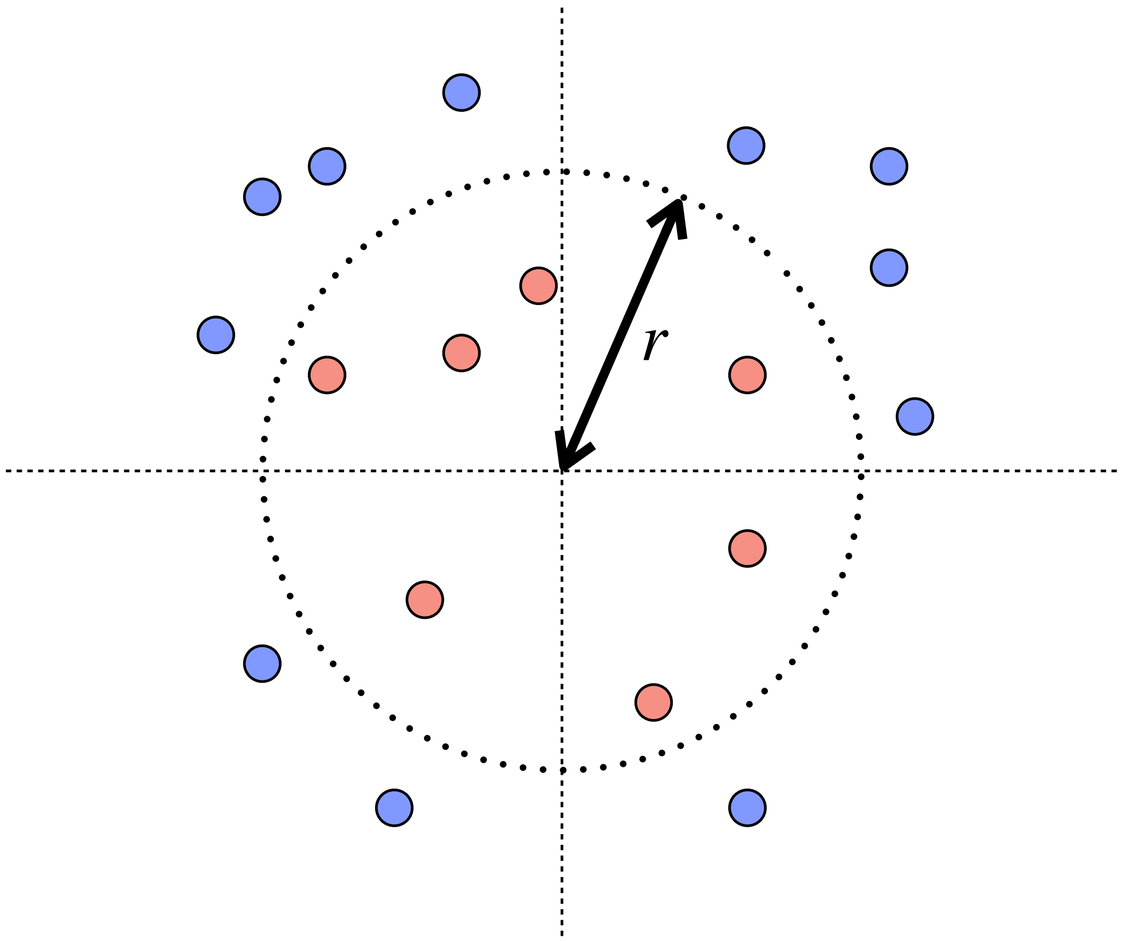
Example Toy Project

- **Goal** - Binary Classification
- **Metric** - Accuracy



Example Toy Project

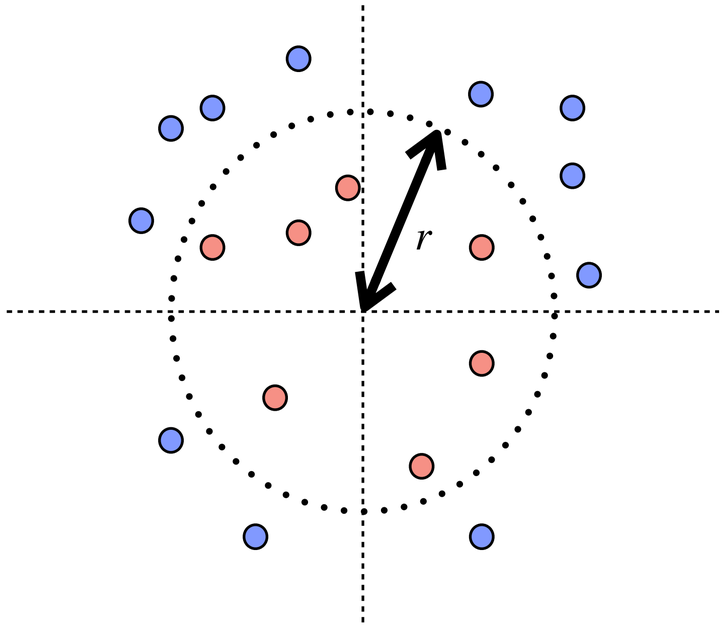
- **Goal** - Binary Classification
- **Metric** - Accuracy
- **Loss** - Binary Cross-Entropy



Example Toy Project

Data

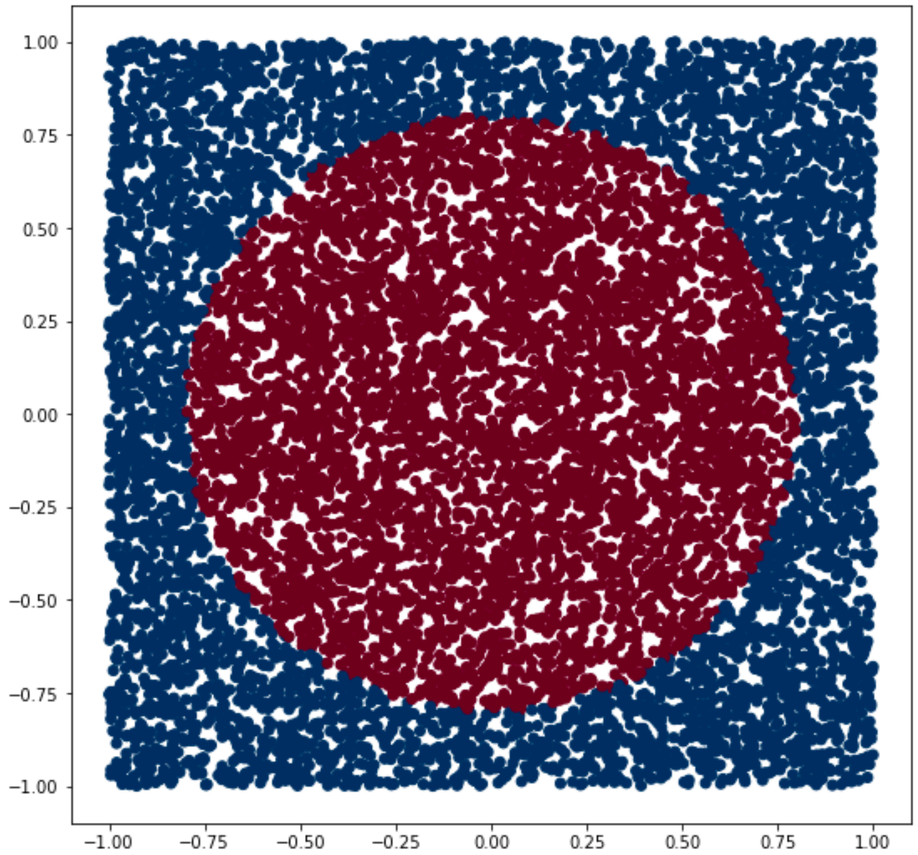
```python
def dataset_Circles(m=10, radius=0.7, noise=0.0):
    # m samples of 2D point
    X = (np.random.rand(2, m) * 2.0) - 1.0
    N = (np.random.rand(2, m)-0.5) * noise
    # compute radius taking noise into account
    R = np.sqrt(np.sum(((X+N)**2), axis=0, keepdims=True))
    Y = (R > radius).astype(float)
    return X, Y
```



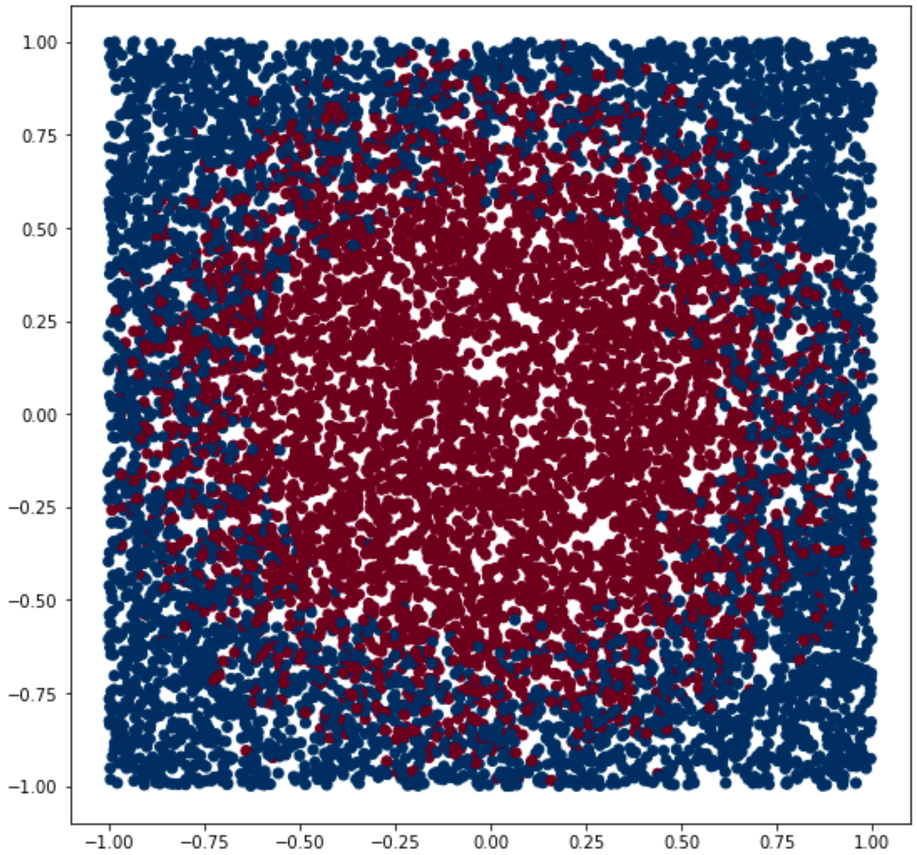
Example Toy Project

Data

```
X, Y = dataset_Circles(m=10000, radius=0.8, noise=0.0)
```



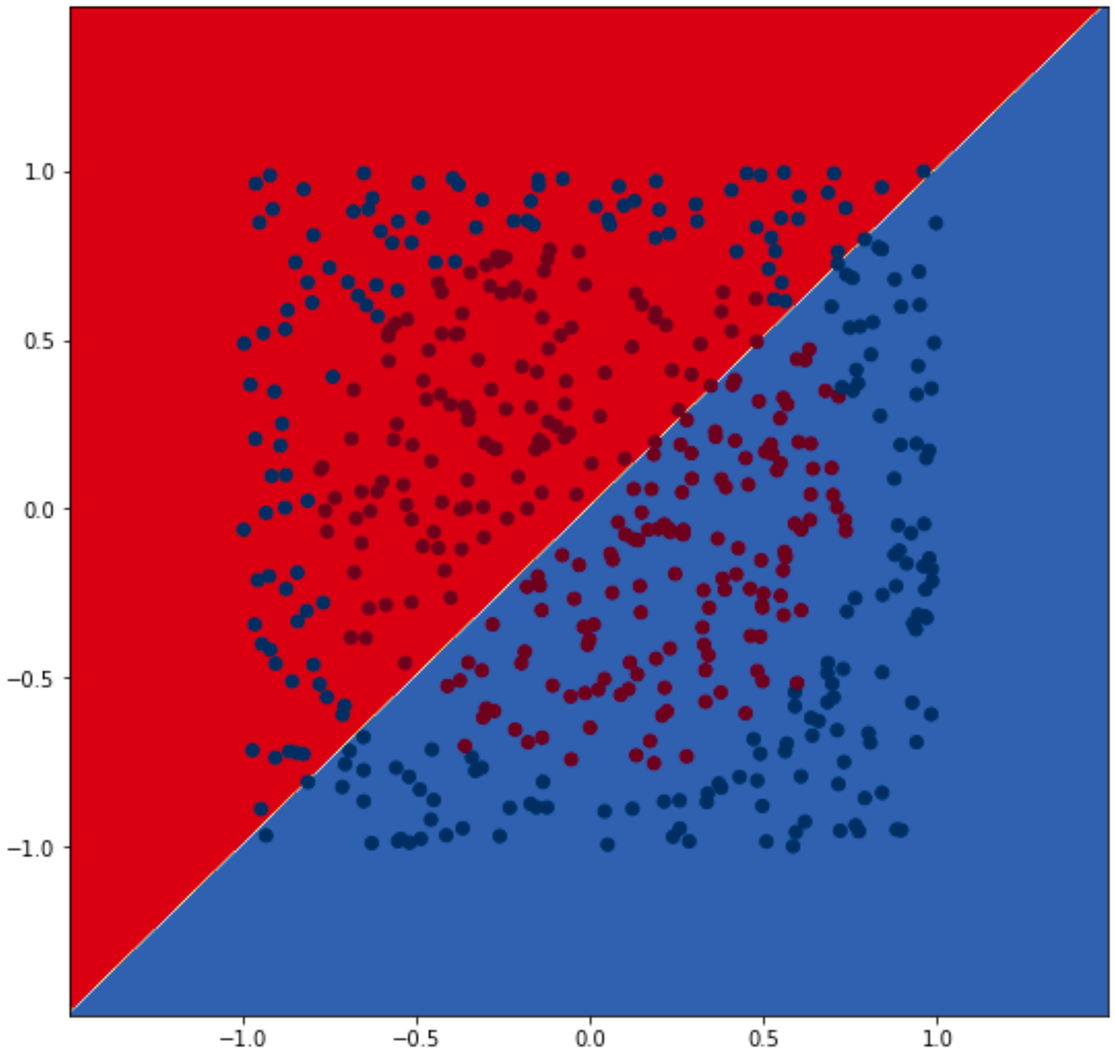

```
X, Y = dataset_Circles(m=10000, radius=0.8, noise=0.5)
```



Example Toy Project

Visualization

```
def testModel(X):
    x1 = X[0, None]
    x2 = X[1, None]

    yhat = (x1 > x2).astype(float)
    return yhat

# Visualize a small dataset
X, Y = dataset_Circles(m=500, radius=0.8, noise=0.0)
draw_DecisionBoundary(X, Y, testModel)
```

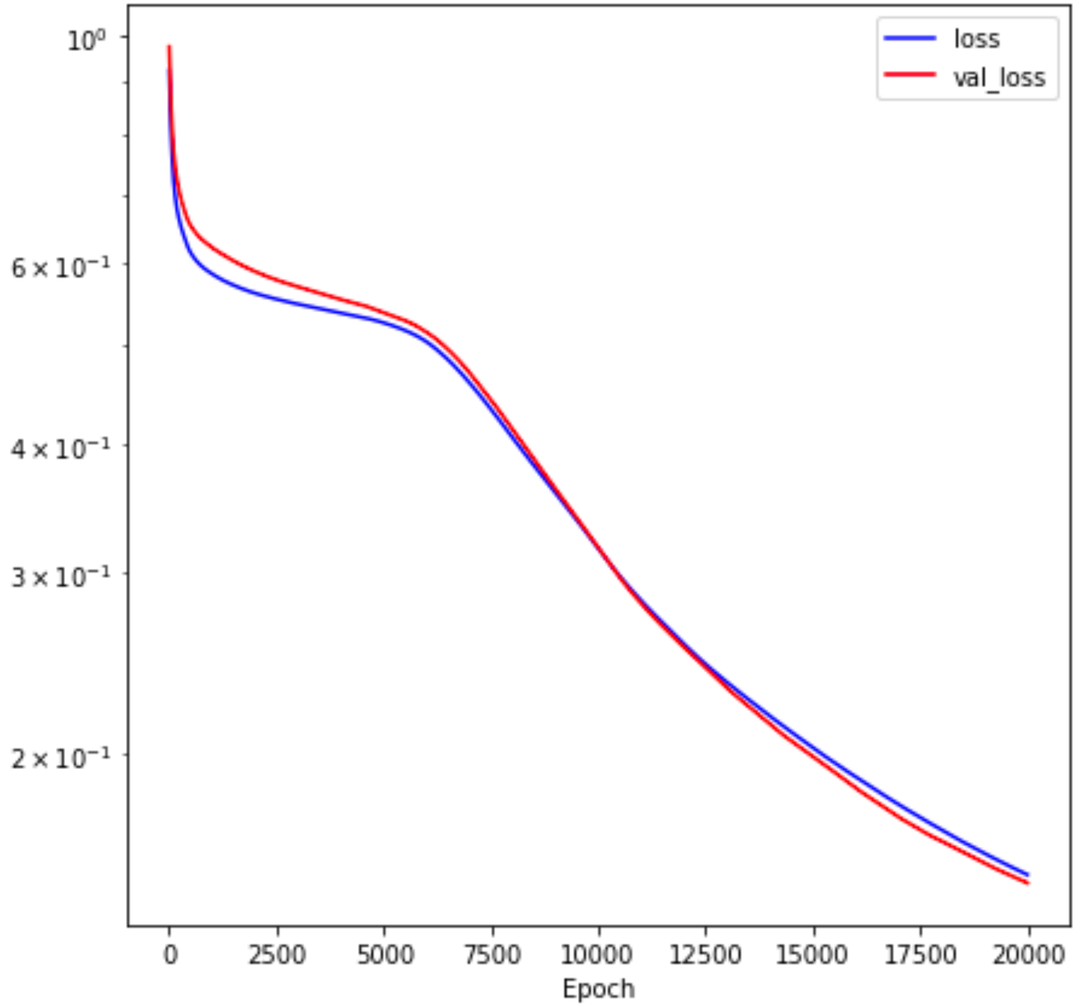


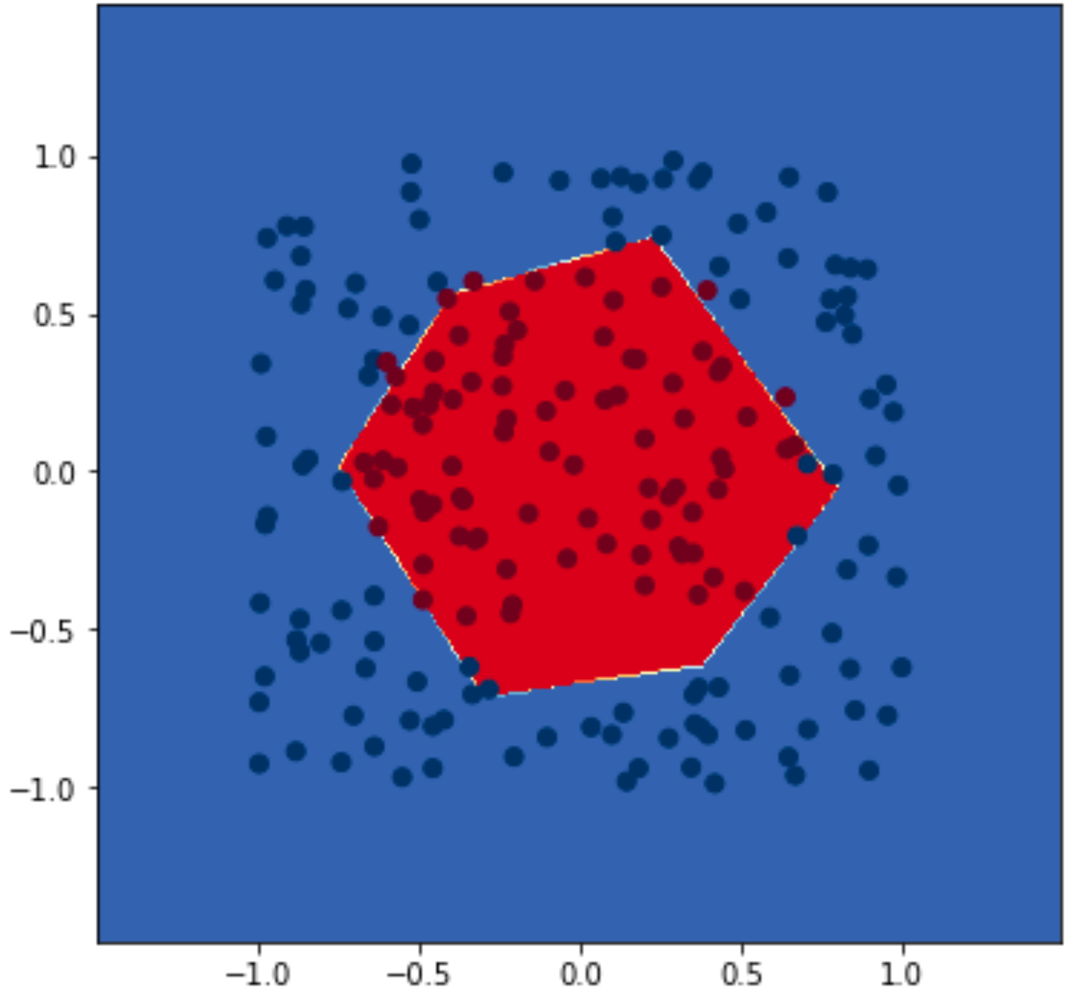
Example Toy Project

- **Goal** - Binary Classification
- **Metric** - Accuracy
- **Loss** - Binary Cross-Entropy
- **Model Architecture** - MLP
  - 1 hidden layer - 6 units
  - 1 output layer - 1 unit
- Learning rate = 0.01

```python
# Training and Validation Dataset
X, Y = dataset_Circles(m=1000)
devX, devY = dataset_Circles(m=200)

nx, m = X.shape
ny, _ = Y.shape

# Model architecture
model = Model([
    Input(nx),
    Dense(6, act='relu'),
    Dense(ny, act='sigmoid')
])

# Train with binary cross-entropy loss
model.initialize(
    loss=CreateLossFunction('bce'),
    optimizer=GradientDescent(0.01)
)

# Train for 20,000 epochs
results = model.train(X, Y, 20000, 0, devX, devY)

# Visualize
draw_TrainingResults(results)
draw_DecisionBoundary(devX, devY, model)
```

Example Toy Project

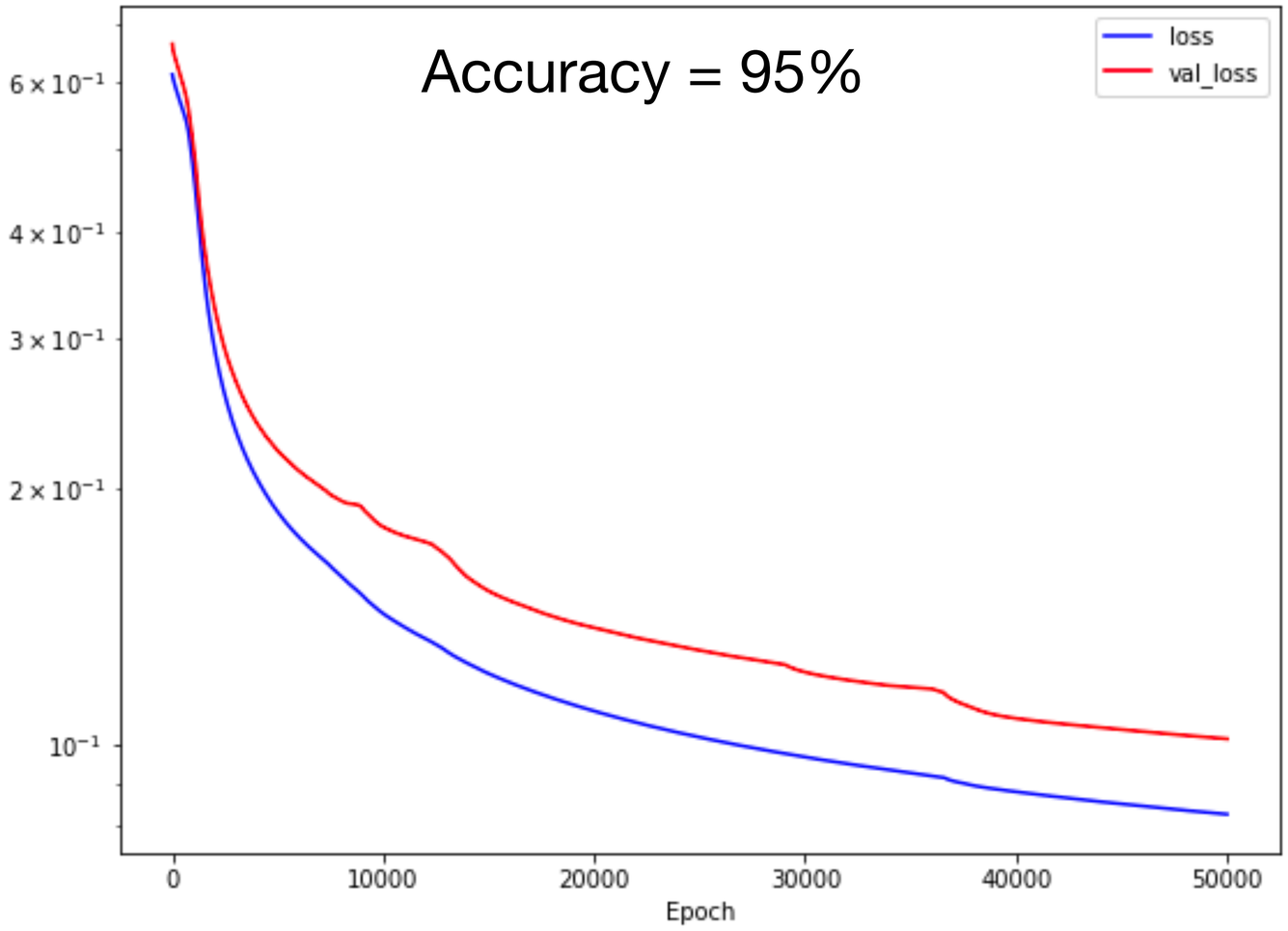


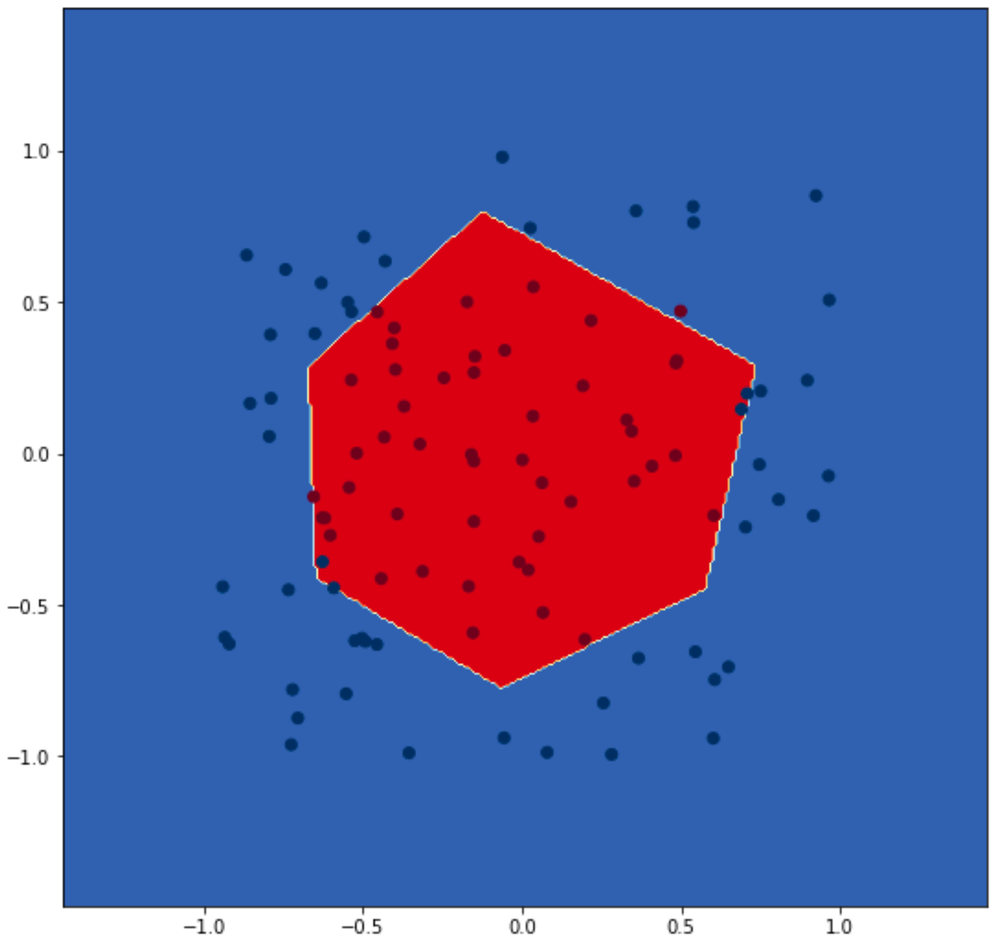


Example Toy Project

Underfit

```
layers = [
    Input(2),
    Dense(3, act='relu'),
    Dense(1, act='sigmoid')
]
# Train underfit scenario
test_UnderfitOverfit(mTrain=500, mDev=100, noise=0.0, layers=layers)
```

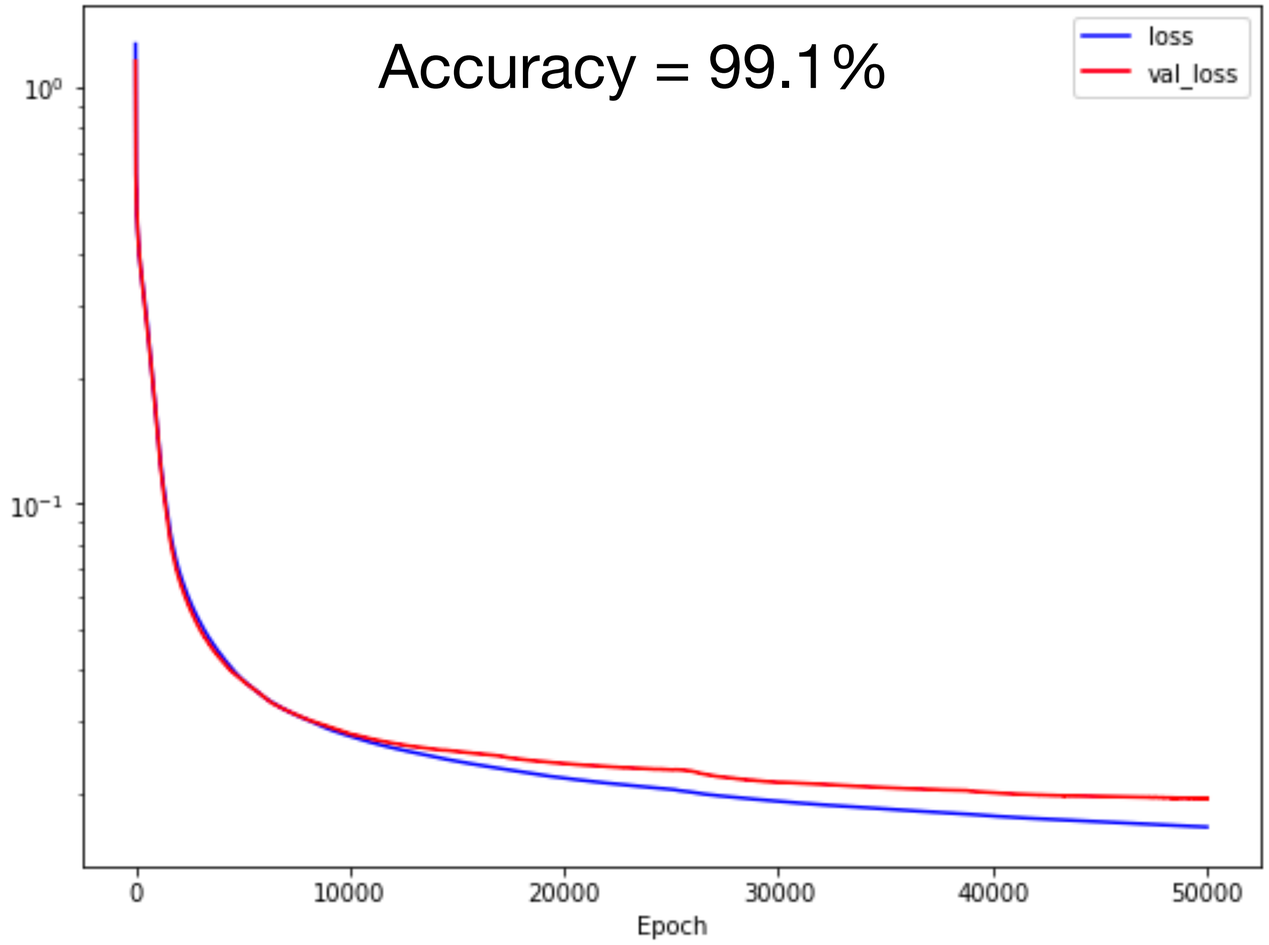


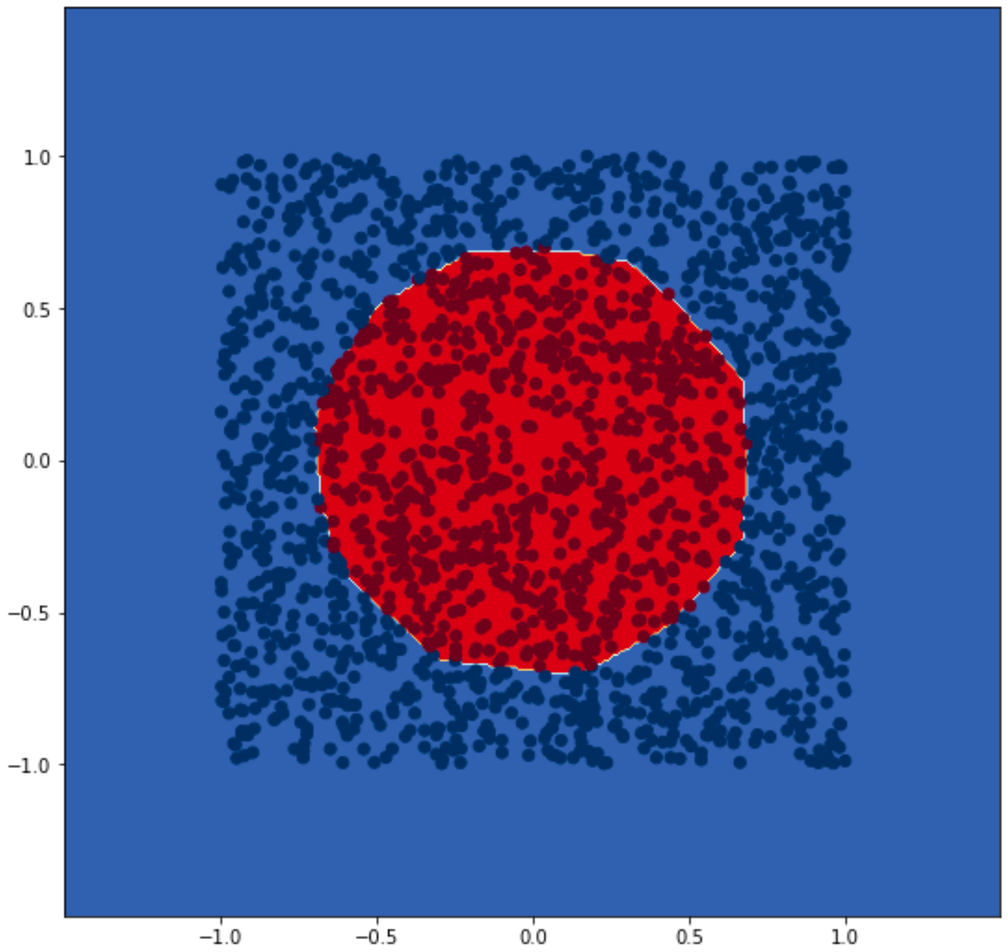


Example Toy Project

“Just Right”

```
layers = [
    Input(2),
    Dense(8, act='relu'),
    Dense(8, act='relu'),
    Dense(1, act='sigmoid')
]
# Train just-right scenario
test_UnderfitOverfit(mTrain=5000, mDev=2000, noise=0.0, layers=layers)
```

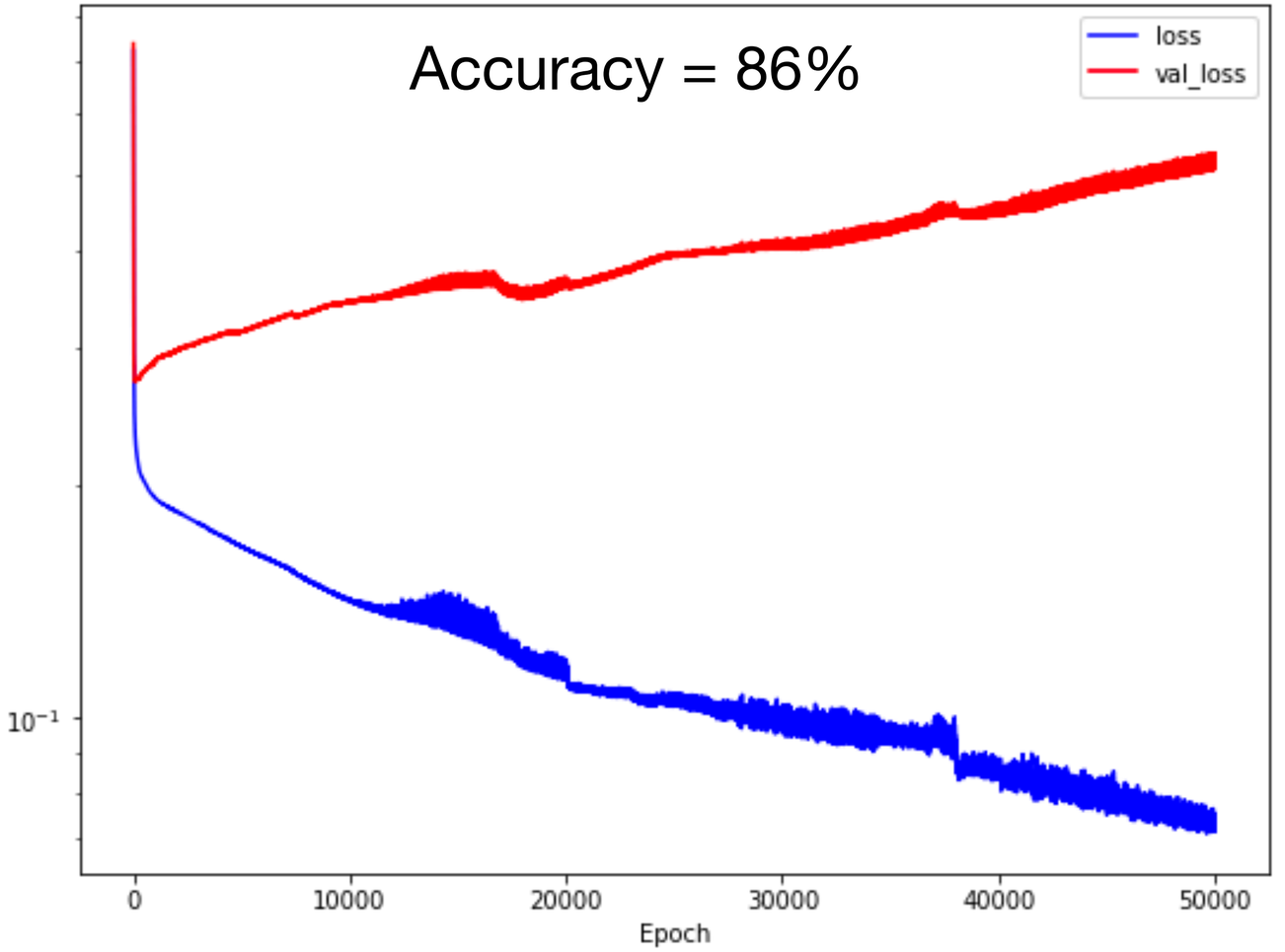


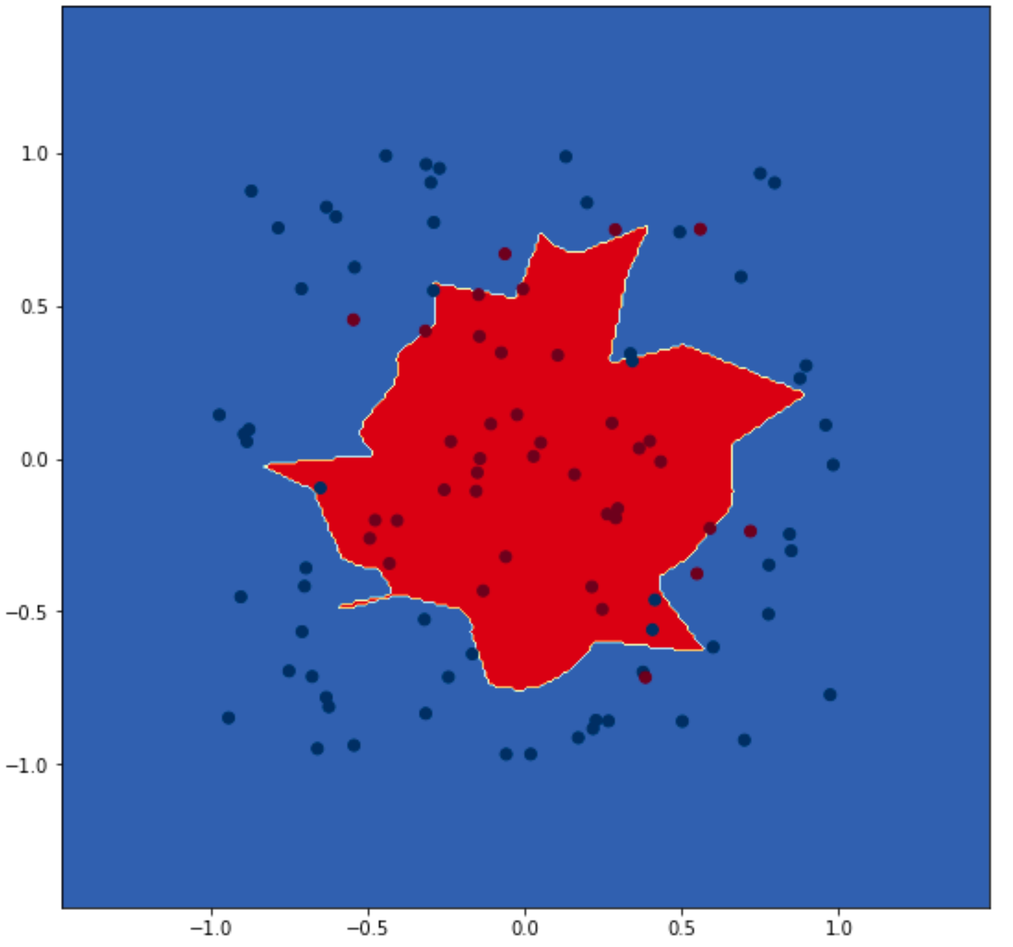


Example Toy Project

Overfit

```
layers = [
    Input(2),
    Dense(20, act='relu'),
    Dense(20, act='relu'),
    Dense(1, act='sigmoid')
]
# Train overfit scenario
test_UnderfitOverfit(mTrain=200, mDev=100, noise=0.5, layers=layers)
```

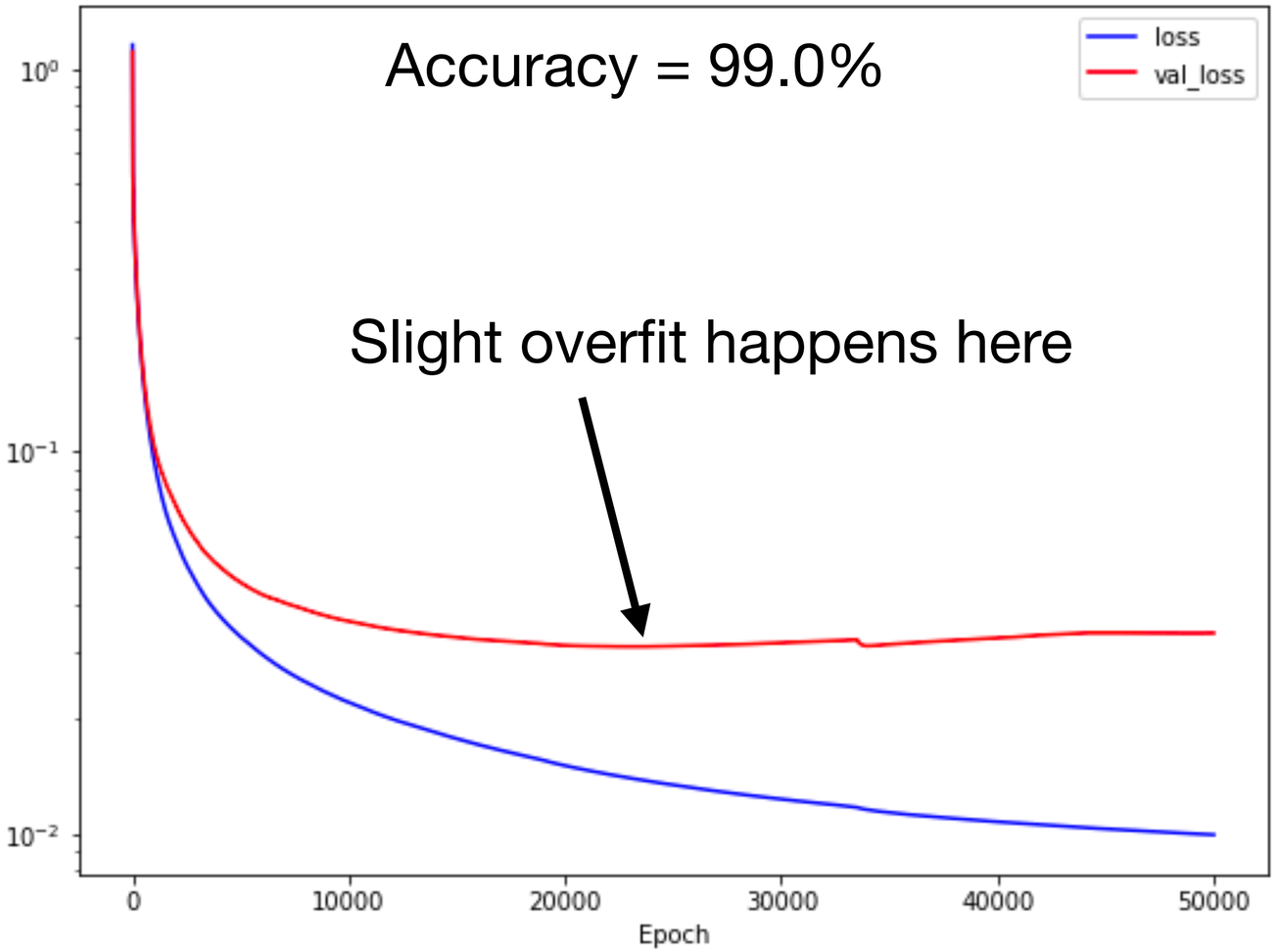


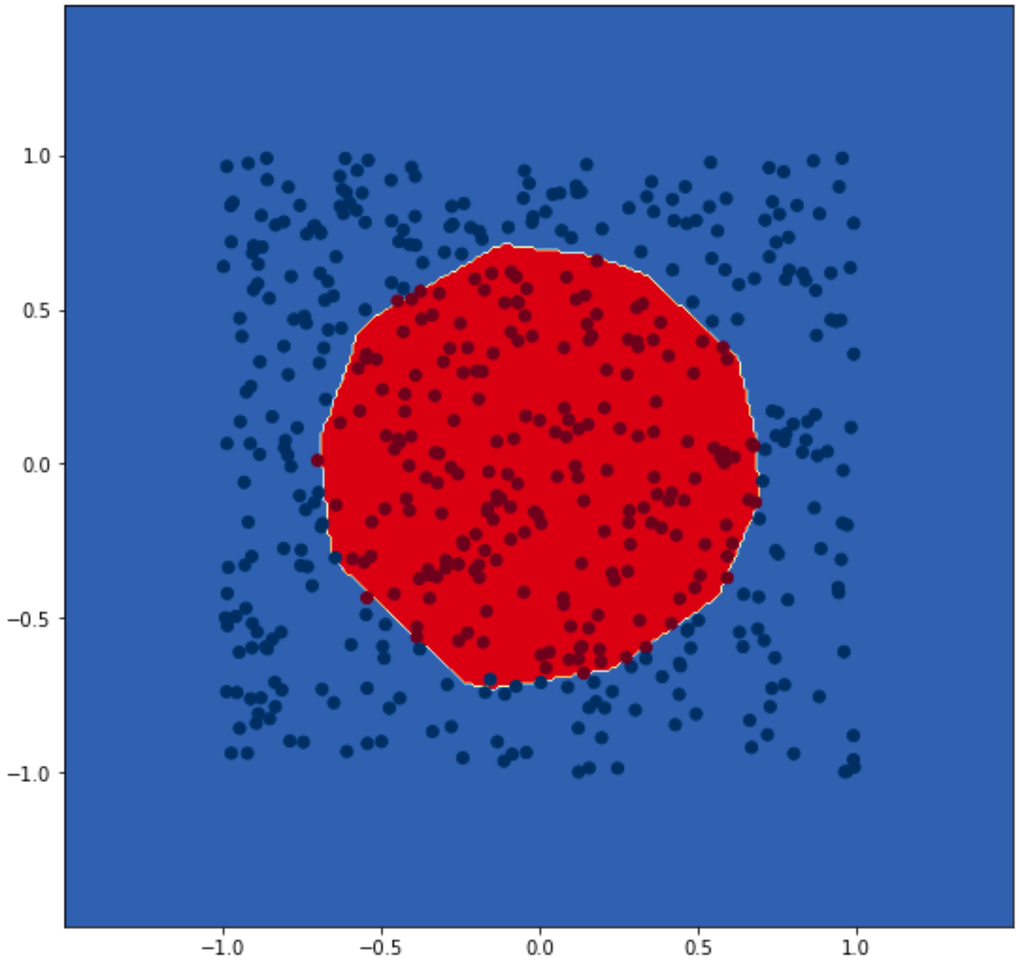


Example Toy Project

“Just Right 2”

```
layers = [
    Input(2),
    Dense(10, act='relu'),      # 2 more neurons in hidden
    Dense(10, act='relu'),      # layers
    Dense(1, act='sigmoid')
]
# Train Overfit-2 - slightly smaller dataset
test_UnderfitOverfit(mTrain=1000, mDev=500, noise=0.0, layers=layers)
```



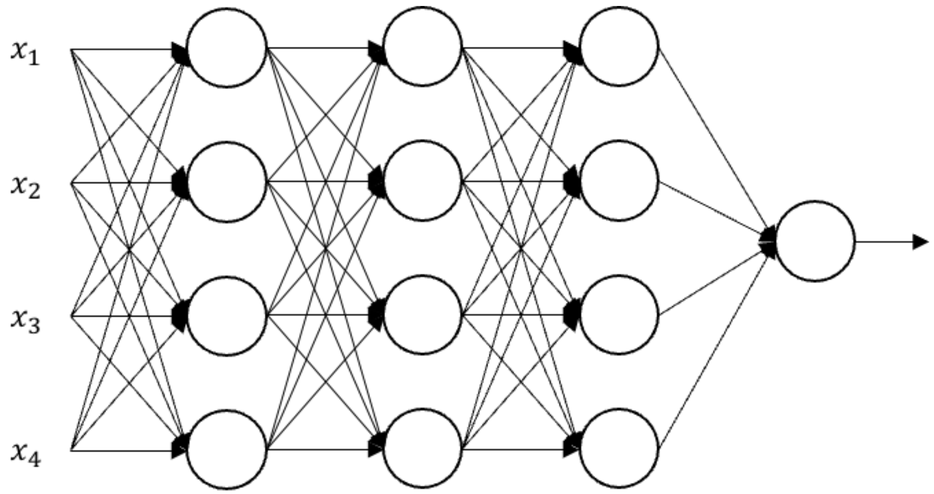


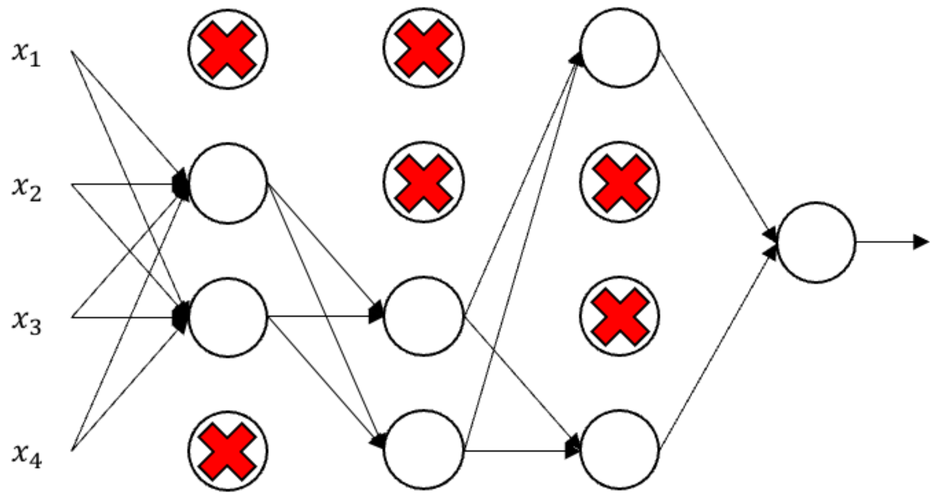
Regularization

- **Prevent overfitting**
- Improve the generalization performance of a model
- Discourage the model from becoming too complex
- Control the model’s ability to fit noise in the training data

Regularization

DropOut





50 % chance, that a neuron will be used

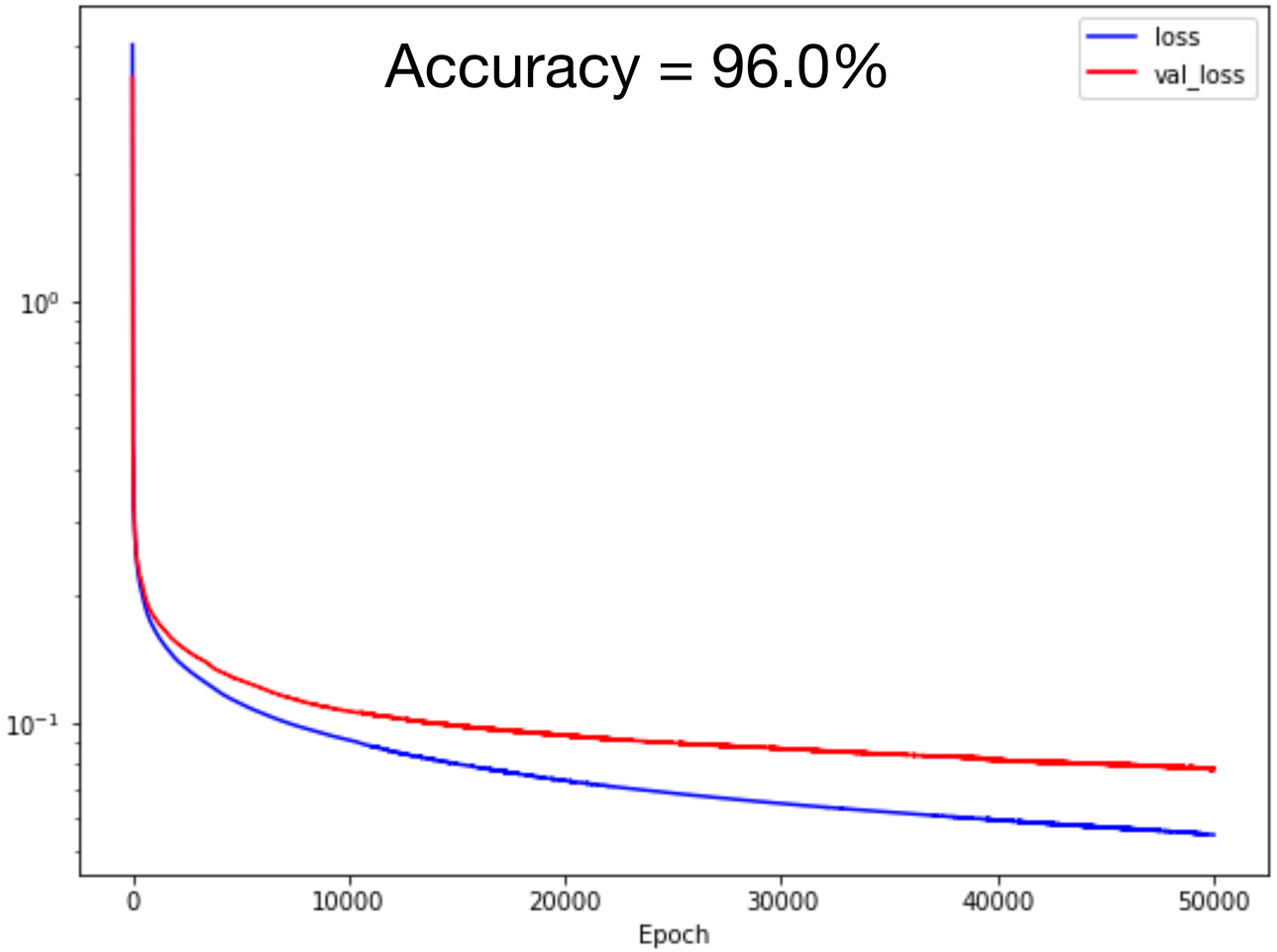
Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting." *The journal of machine learning research* 15.1 (2014): 1929-1958.

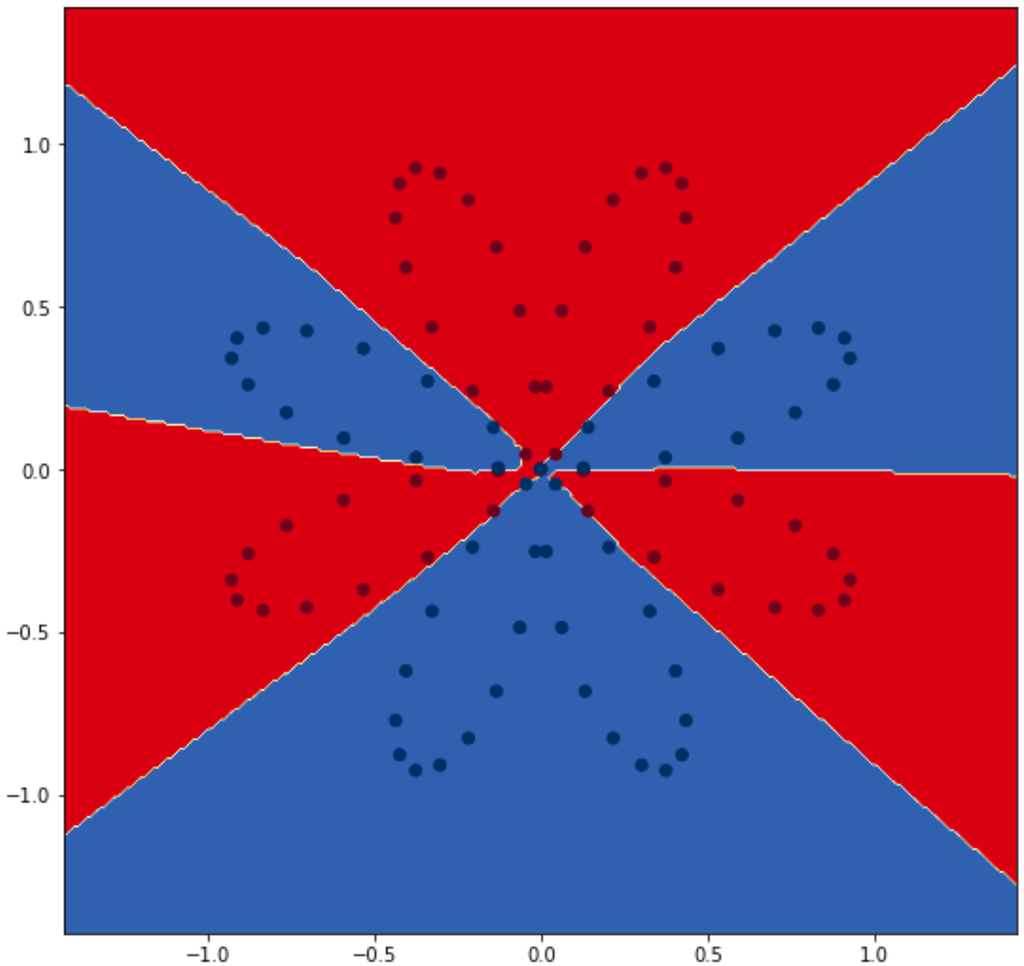
Regularization

DropOut

```
layers = [
    Input(2),
    Dense(15, act='relu'),
    Dense(15, act='relu'),
    Dense(1, act='sigmoid')
]
test_DropOutCase(mTrain=500, mDev=100, noise=0.0, layers=layers)
```

No random noise



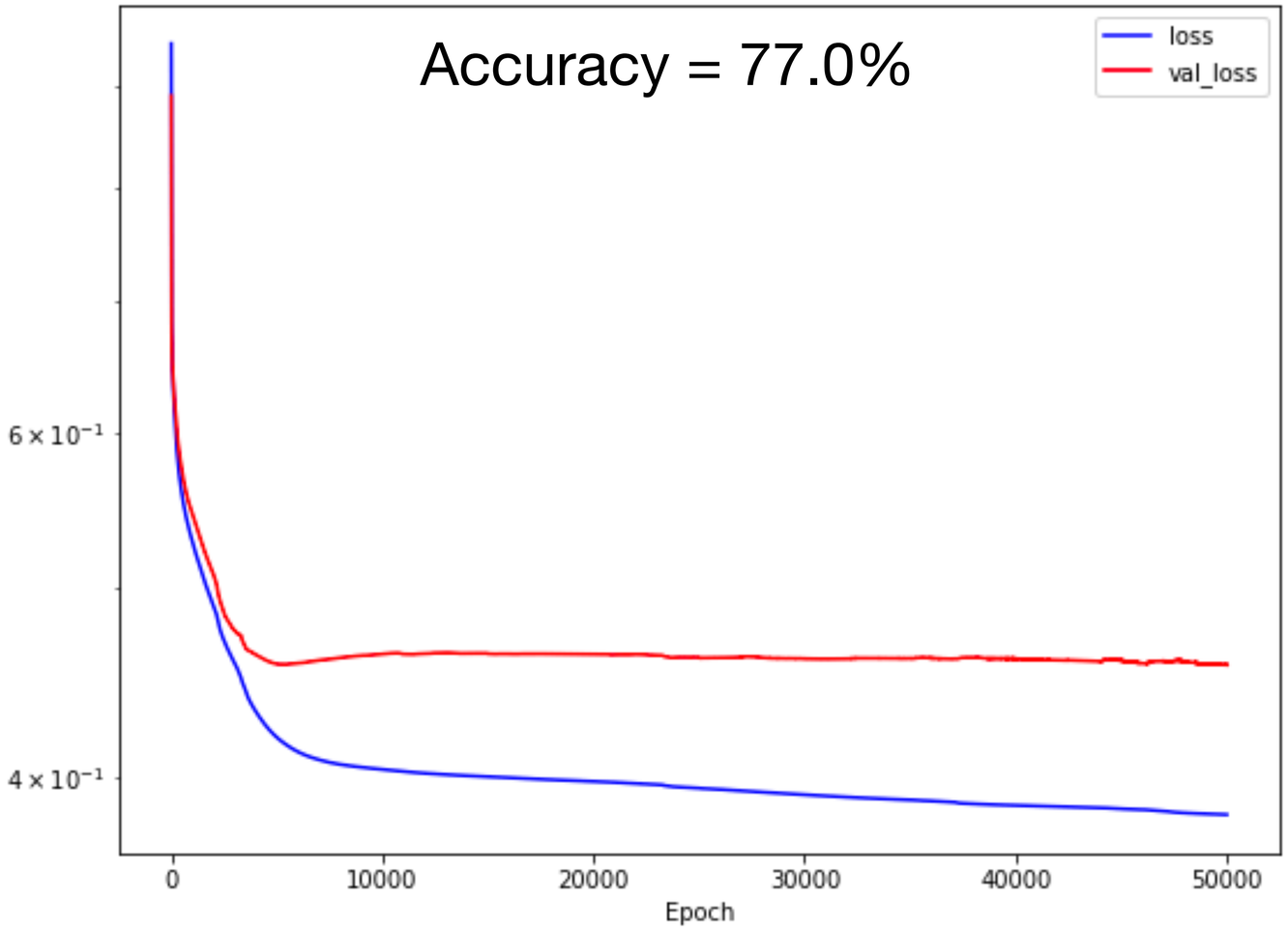


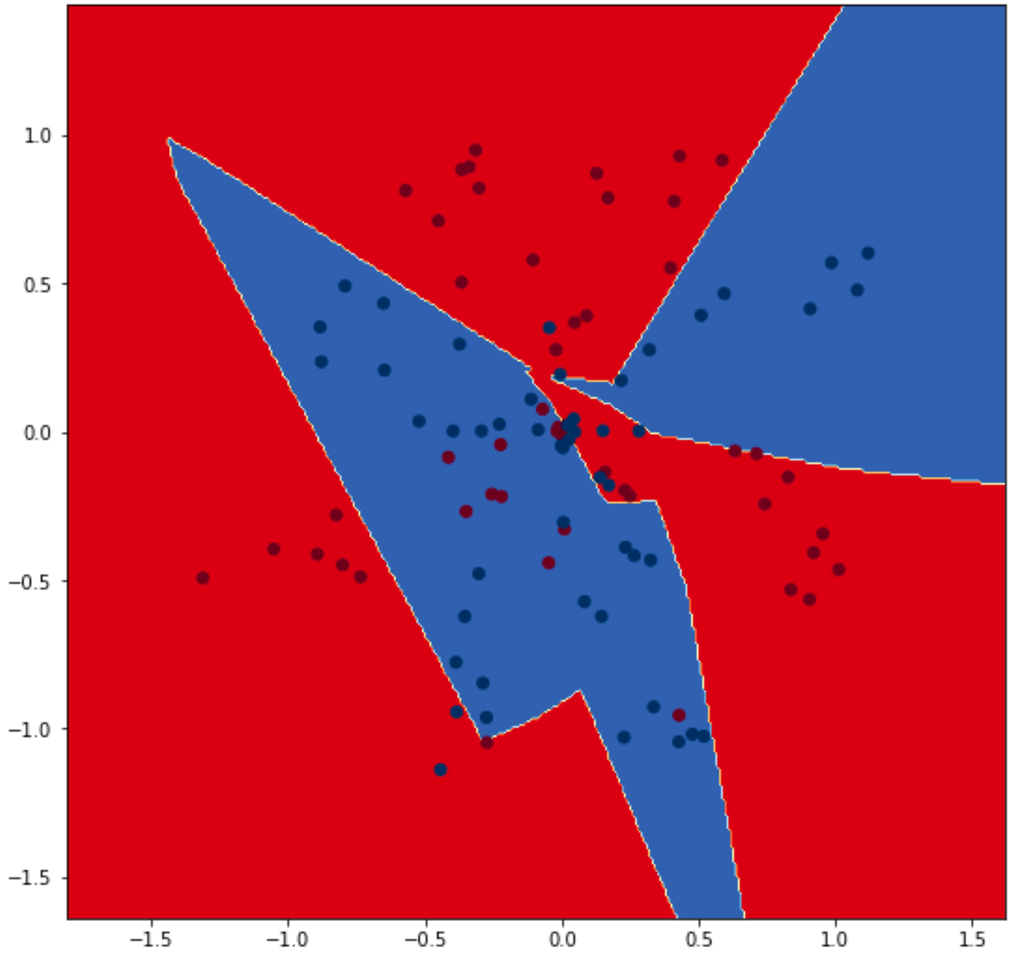
Regularization

DropOut - Poor Fit

```
layers = [
    Input(2),
    Dense(5, act='relu'),
    Dense(5, act='relu'),
    Dense(1, act='sigmoid')
]
test_DropOutCase(mTrain=500, mDev=100, noise=0.2, layers=layers)
```

Noise !!



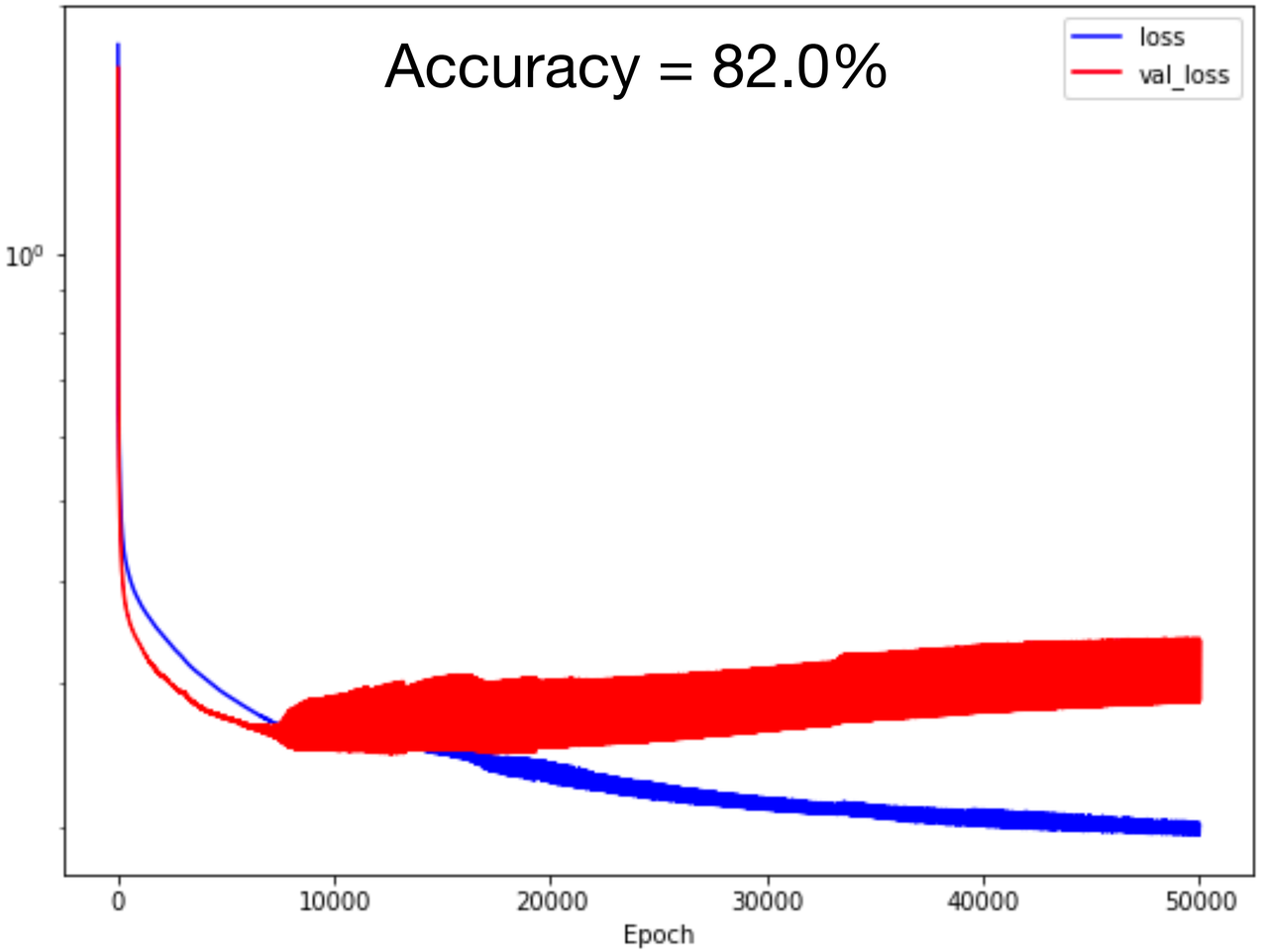


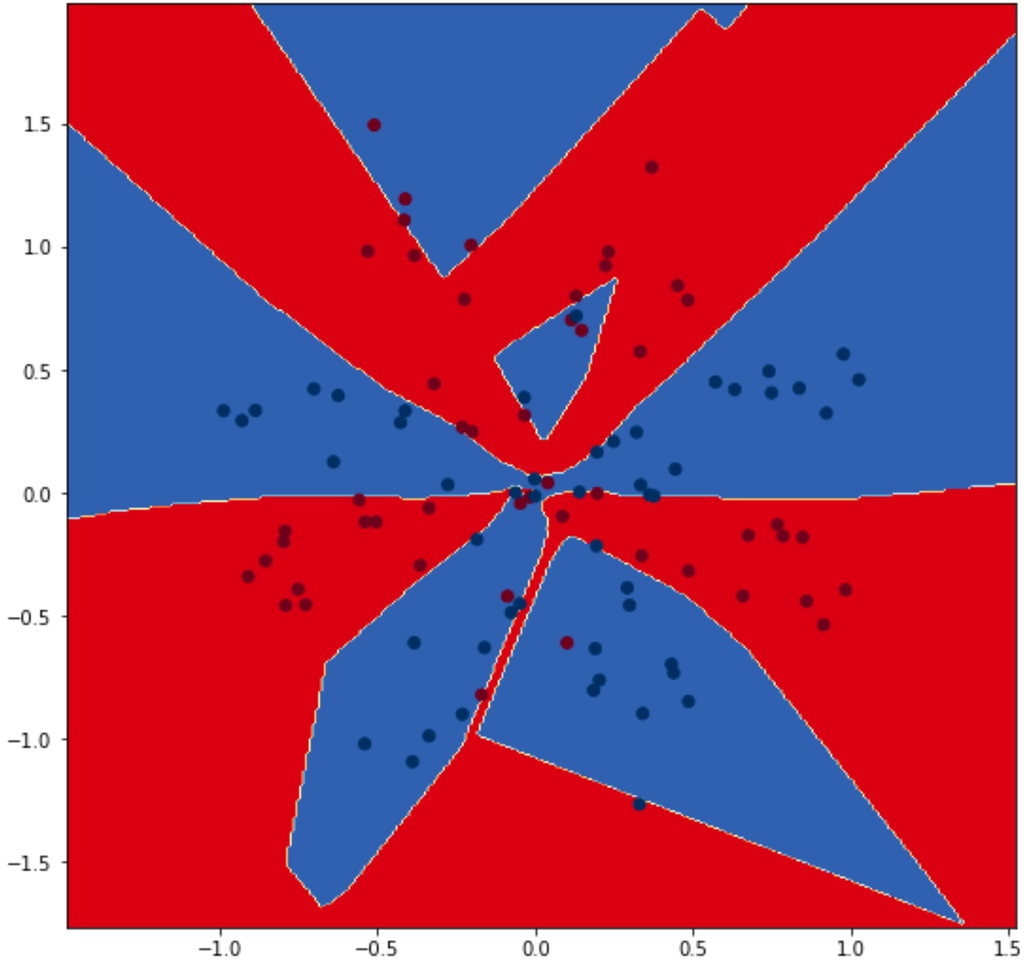
Regularization

DropOut - Stronger Model

```
layers = [
    Input(2),
    Dense(15, act='relu'),
    Dense(15, act='relu'),
    Dense(1, act='sigmoid')
]
test_DropOutCase(mTrain=400, mDev=100, noise=0.2, layers=layers)
```

Noise !!



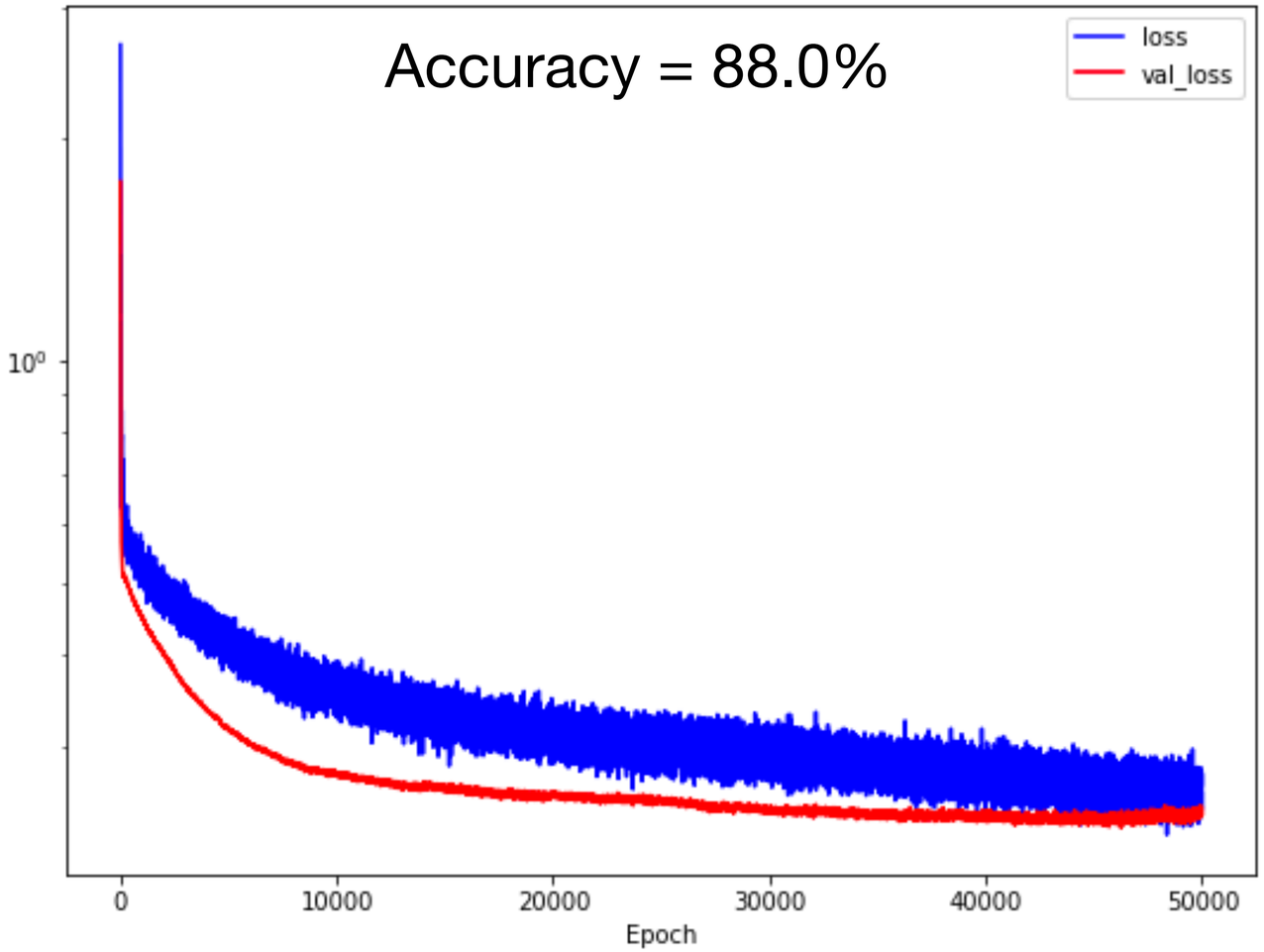


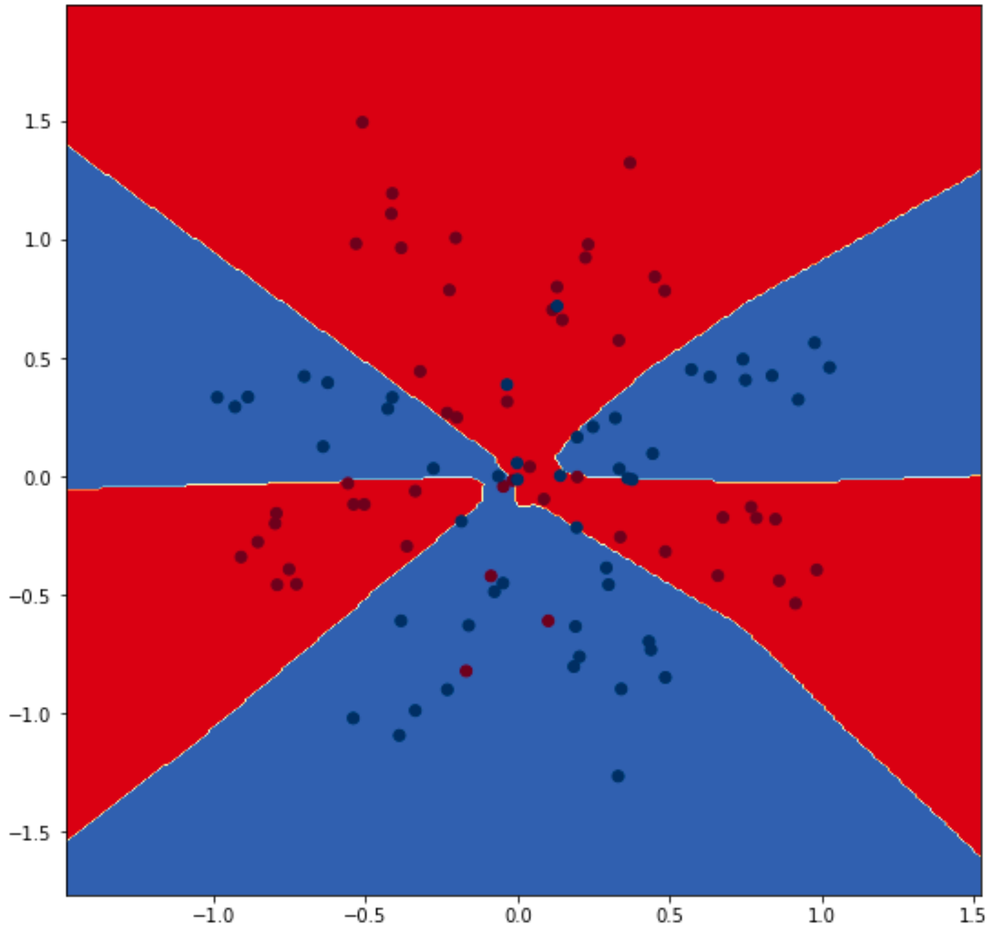
Regularization

DropOut - Stronger Model

```
layers = [
    Input(2),
    Dense(15, act='relu'),
    Dense(15, act='relu'),
    DropOut(0.4),
    Dense(1, act='sigmoid')
]
test_DropOutCase(mTrain=400, mDev=100, noise=0.2, layers=layers)
```

DropOut !!





Regularization

L1, L2

- Large values of weights - enables model to fit noise in training data
- We want smaller weights

$$\widetilde{J}(\Theta, \mathscr{X}) = J(\Theta, \mathscr{X}) + \lambda\Omega(\Theta)$$

λ - regularization weight

$\Omega(\Theta)$ - penalty function

Regularization

L1, L2

- Large values of weights - enables model to fit noise in training data
- We want smaller weights

$$\widetilde{J}(\Theta, \mathscr{X}) = J(\Theta, \mathscr{X}) + \lambda\Omega(\Theta)$$

L1 : $\Omega(\Theta) = \sum_i |w_i|$

L2 : $\Omega(\Theta) = \sum_i w_i^2$

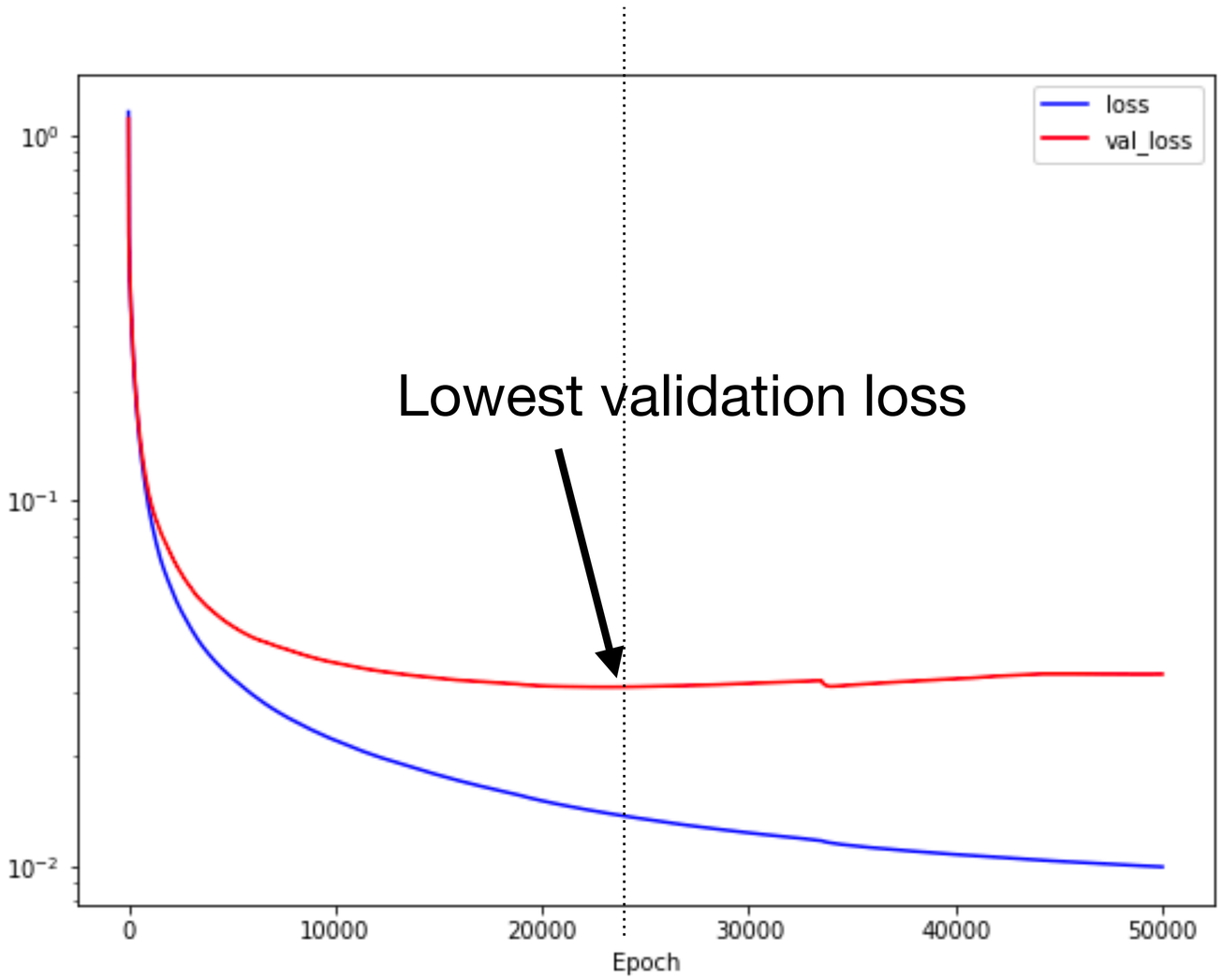
λ - regularization weight

$\Omega(\Theta)$ - penalty function

Regularization

Early Stopping

- Complex model is more likely to overfit
- Training loss is decreasing
- Validation loss is increasing
- Use the model with lowest validation loss



Initialization

Vanishing & Exploding Gradient Problem

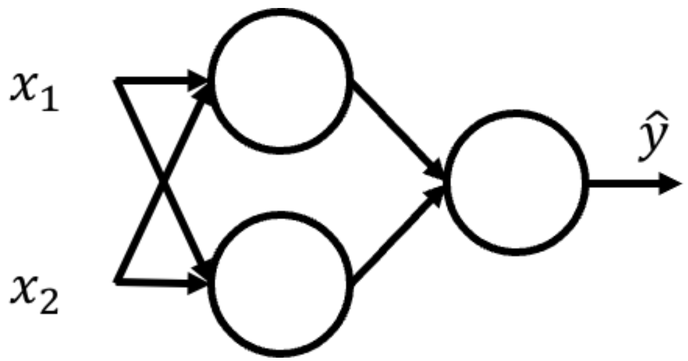
- Consider the following example

Bengio, Yoshua, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult." *IEEE transactions on neural networks* 5.2 (1994): 157-166.

Initialization

Vanishing & Exploding Gradient Problem

- Consider the following example
  - ignore bias, only use linear activation function



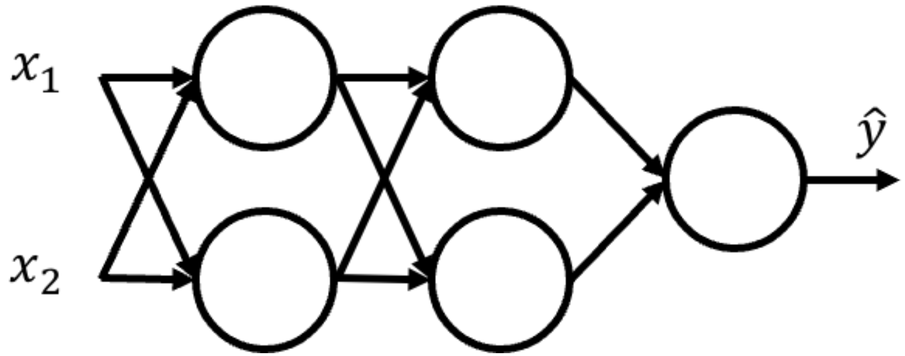
$$\hat{y} = W^{[2]}W^{[1]}x$$

Bengio, Yoshua, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult." *IEEE transactions on neural networks* 5.2 (1994): 157-166.

Initialization

Vanishing & Exploding Gradient Problem

- Consider the following example
  - ignore bias, only use linear activation function

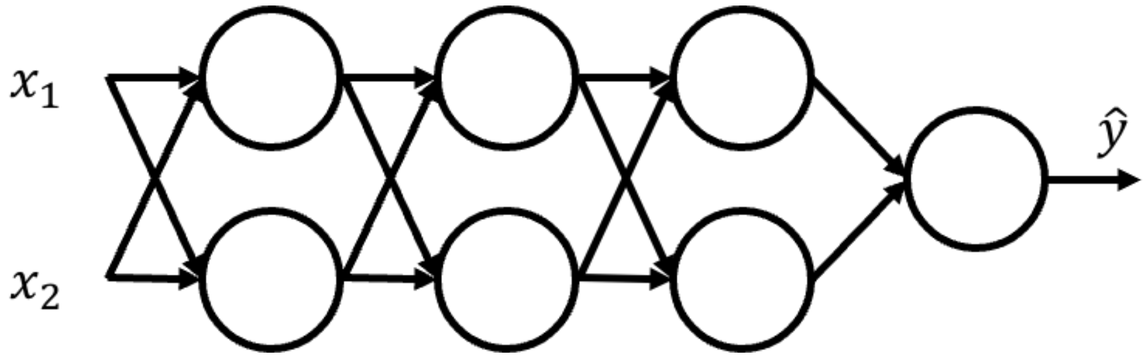


$$\hat{y} = W^{[3]}W^{[2]}W^{[1]}x$$

Initialization

Vanishing & Exploding Gradient Problem

- Consider the following example
  - ignore bias, only use linear activation function

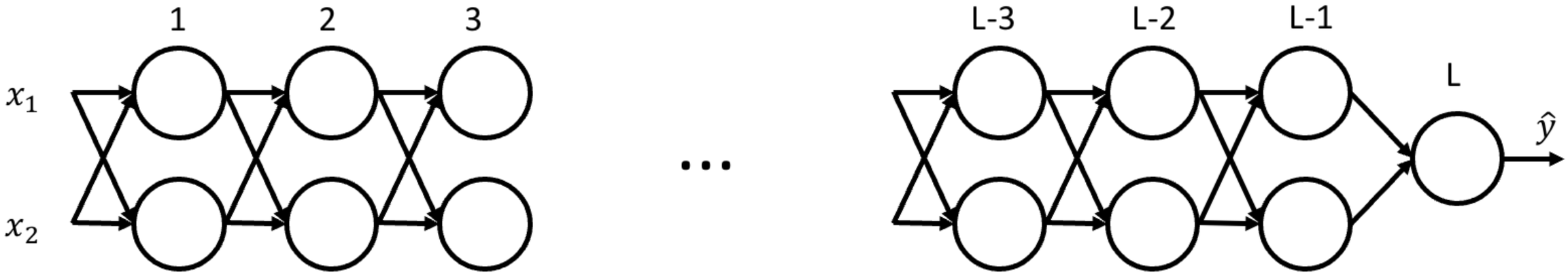


$$\hat{y} = W^{[4]}W^{[3]}W^{[2]}W^{[1]}x$$

Initialization

Vanishing & Exploding Gradient Problem

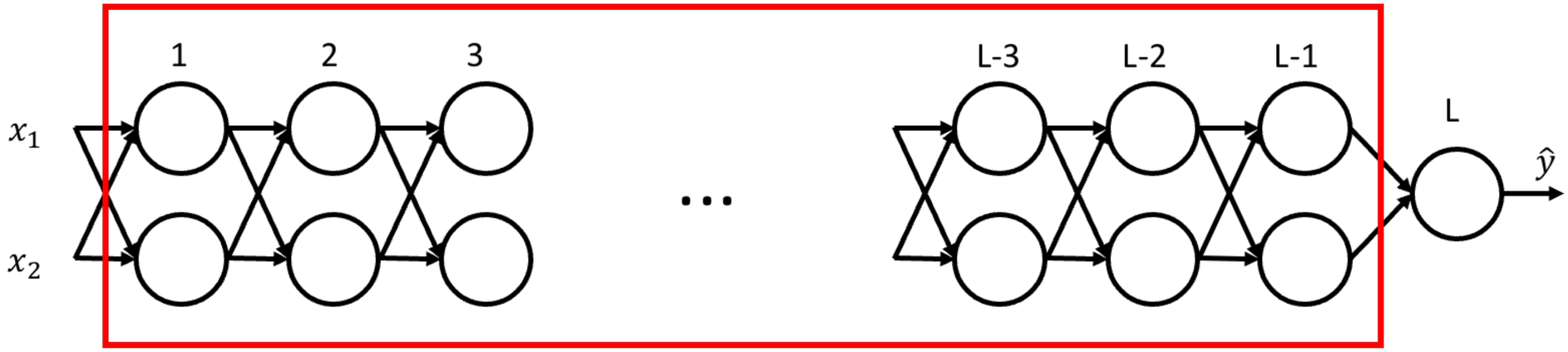
- Consider the following example
  - ignore bias, only use linear activation function



$$\hat{y} = W^{[L]} \ldots W^{[4]} W^{[3]} W^{[2]} W^{[1]} x$$

Initialization

Vanishing & Exploding Gradient Problem



$L = 21$

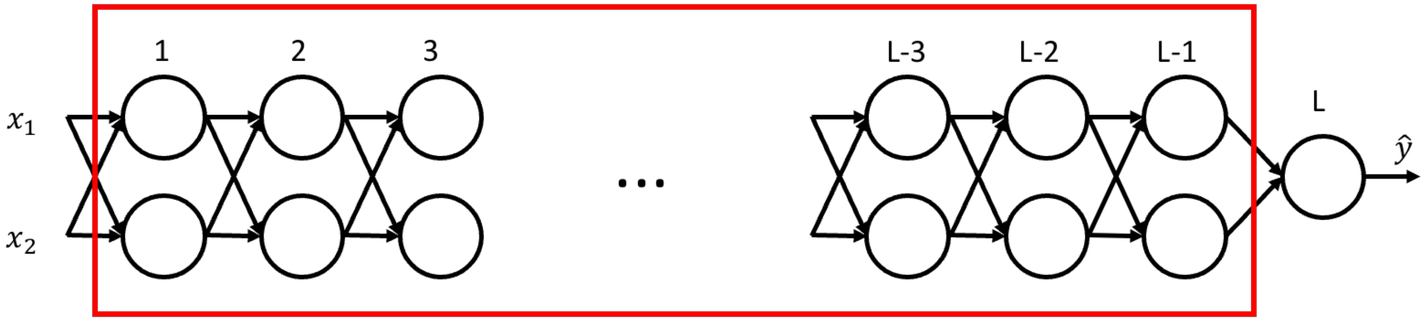
$$\hat{y} = W^{[L]} \ldots W^{[4]} W^{[3]} W^{[2]} W^{[1]} x$$

$$\hat{y} = W^{[L]} W^{L-1} x$$

W is raised to the number of layers

Initialization

Vanishing & Exploding Gradient Problem



$$L = 21$$

$$\hat{y} = W^{[L]} \ldots W^{[4]} W^{[3]} W^{[2]} W^{[1]} x$$

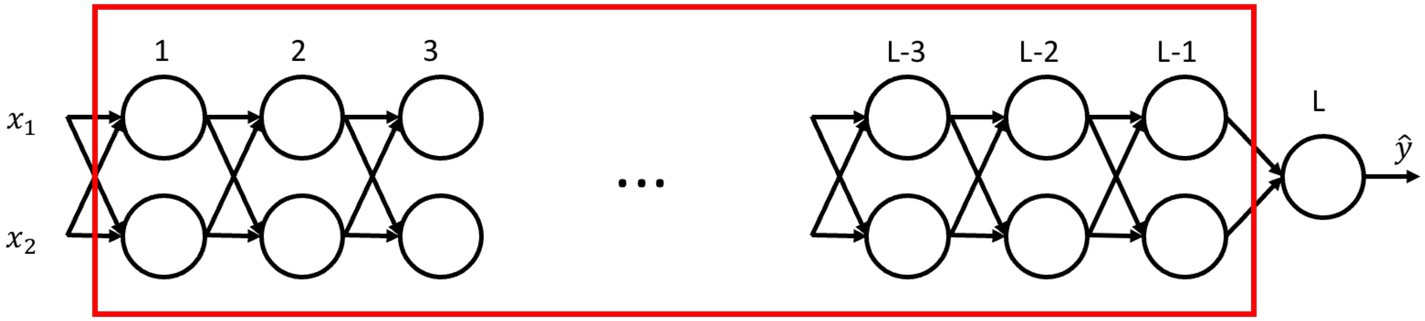
$$\hat{y} = W^{[L]} W^{L-1} x$$

$$W = \begin{pmatrix} 1.5 & 0 \\ 0 & 1.5 \end{pmatrix}$$

W is raised to the number of layers

Initialization

Vanishing & Exploding Gradient Problem



$L = 21$

$$\hat{y} = W^{[L]} \ldots W^{[4]} W^{[3]} W^{[2]} W^{[1]} x$$

$$\hat{y} = W^{[L]} W^{L-1} x$$

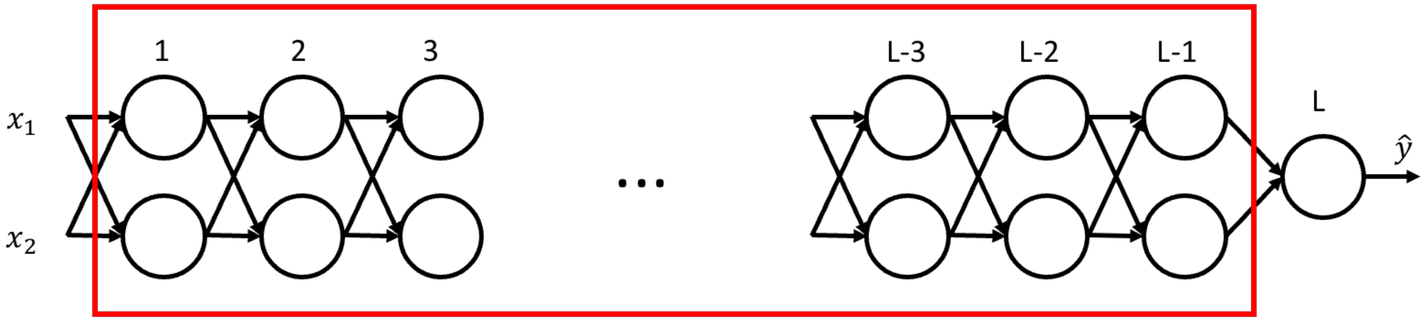
W is raised to the number of layers

$$W = \begin{pmatrix} 1.5 & 0 \\ 0 & 1.5 \end{pmatrix}$$

$$W^2 = \begin{pmatrix} 2.25 & 0 \\ 0 & 2.25 \end{pmatrix}$$

Initialization

Vanishing & Exploding Gradient Problem



$L = 21$

$$\hat{y} = W^{[L]} \ldots W^{[4]} W^{[3]} W^{[2]} W^{[1]} x$$

$$\hat{y} = W^{[L]} W^{L-1} x$$

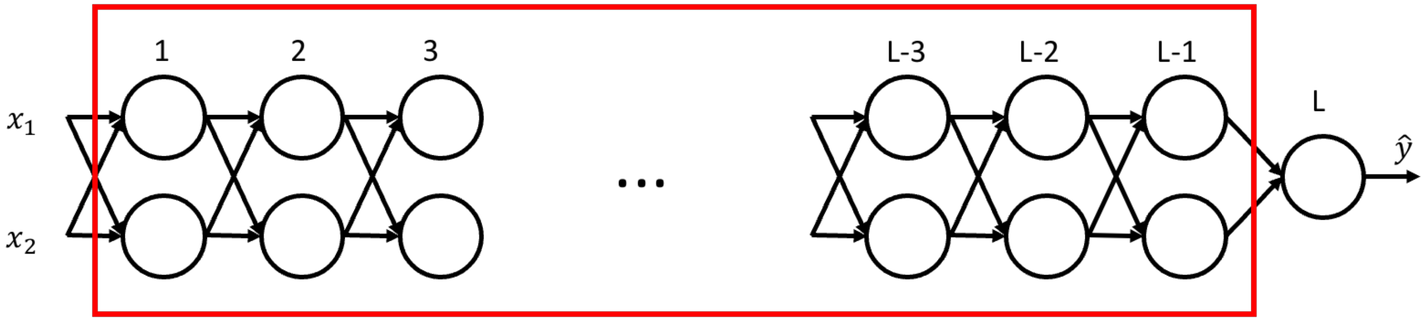
W is raised to the number of layers

$$W = \begin{pmatrix} 1.5 & 0 \\ 0 & 1.5 \end{pmatrix}$$

$$W^3 = \begin{pmatrix} 3.75 & 0 \\ 0 & 3.75 \end{pmatrix}$$

Initialization

Vanishing & Exploding Gradient Problem



$L = 21$

$$\hat{y} = W^{[L]} \ldots W^{[4]} W^{[3]} W^{[2]} W^{[1]} x$$

$$\hat{y} = W^{[L]} W^{L-1} x$$

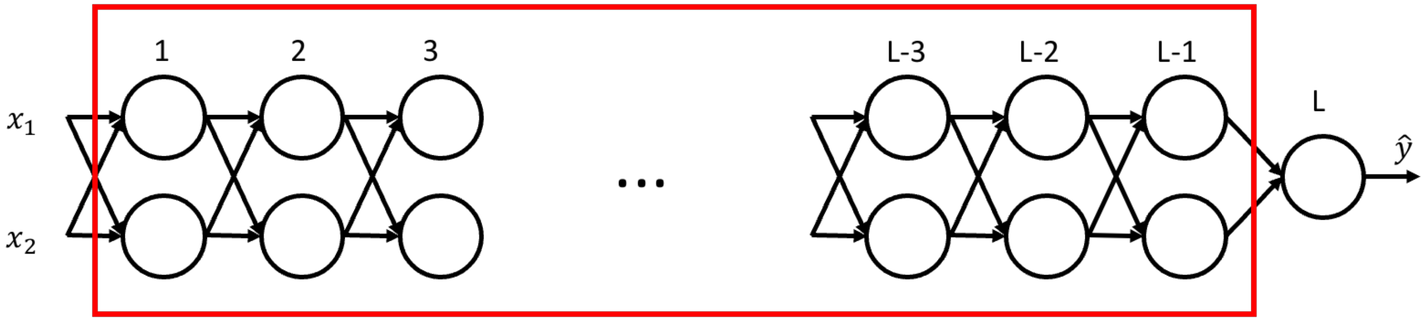
W is raised to the number of layers

$$W = \begin{pmatrix} 1.5 & 0 \\ 0 & 1.5 \end{pmatrix}$$

$$W^{20} = \begin{pmatrix} 3325.26 & 0 \\ 0 & 3325.26 \end{pmatrix}$$

Initialization

Vanishing & Exploding Gradient Problem



$$L = 21$$

$$\hat{y} = W^{[L]} \ldots W^{[4]} W^{[3]} W^{[2]} W^{[1]} x$$

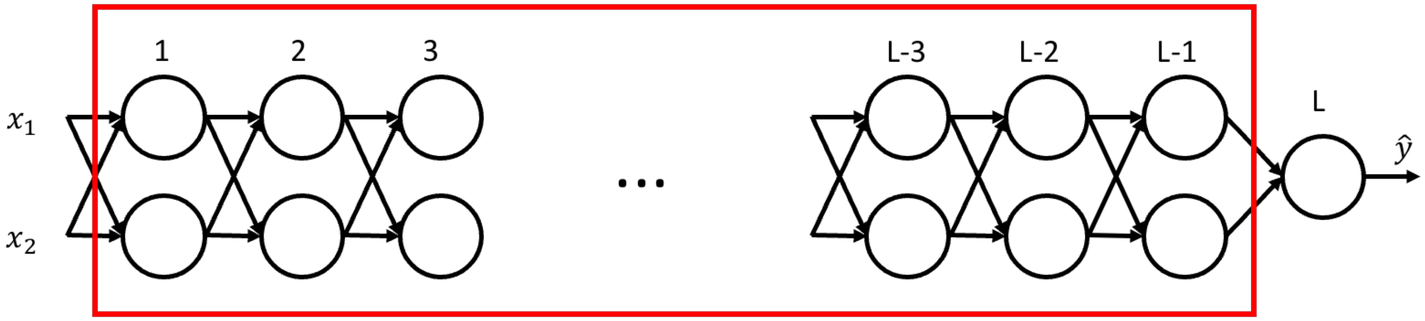
$$\hat{y} = W^{[L]} W^{L-1} x$$

$$W = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix}$$

W is raised to the number of layers

Initialization

Vanishing & Exploding Gradient Problem



$L = 21$

$$\hat{y} = W^{[L]} \ldots W^{[4]} W^{[3]} W^{[2]} W^{[1]} x$$

$$\hat{y} = W^{[L]} W^{L-1} x$$

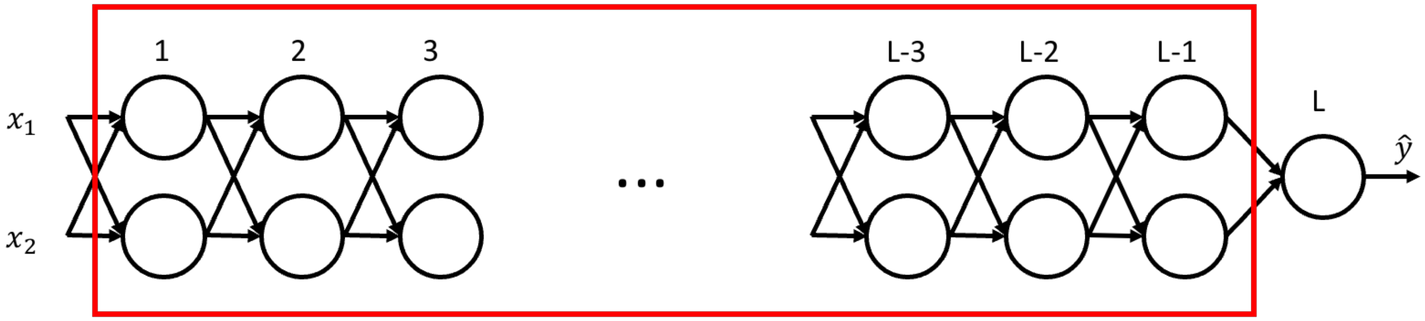
W is raised to the number of layers

$$W = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix}$$

$$W^2 = \begin{pmatrix} 0.25 & 0 \\ 0 & 0.25 \end{pmatrix}$$

Initialization

Vanishing & Exploding Gradient Problem



$L = 21$

$$\hat{y} = W^{[L]} \ldots W^{[4]} W^{[3]} W^{[2]} W^{[1]} x$$

$$\hat{y} = W^{[L]} W^{L-1} x$$

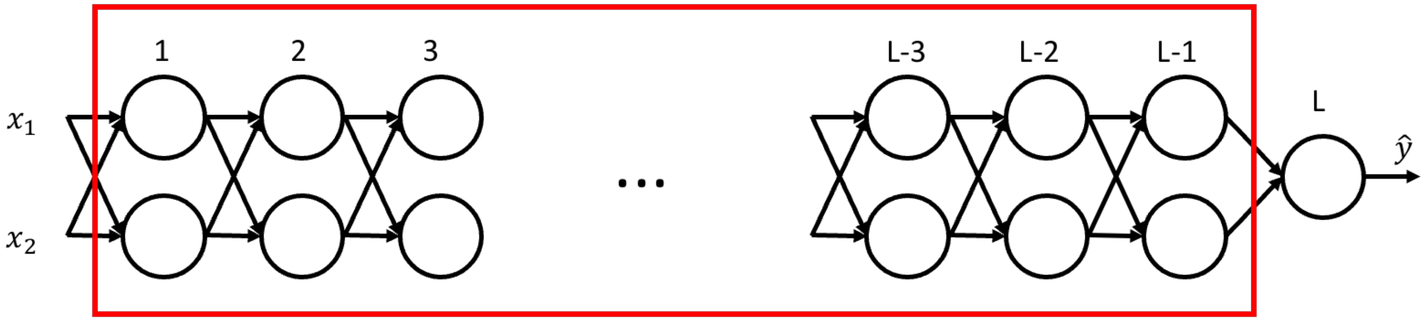
W is raised to the number of layers

$$W = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix}$$

$$W^3 = \begin{pmatrix} 0.125 & 0 \\ 0 & 0.125 \end{pmatrix}$$

Initialization

Vanishing & Exploding Gradient Problem



$L = 21$

$$\hat{y} = W^{[L]} \ldots W^{[4]} W^{[3]} W^{[2]} W^{[1]} x$$

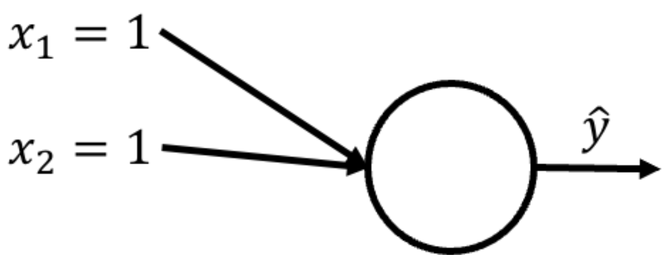
$$\hat{y} = W^{[L]} W^{L-1} x$$

$$W = \begin{pmatrix} 0.5 & 0 \\ 0 & 0.5 \end{pmatrix}$$

W is raised to the number of layers

$$W^{20} = \begin{pmatrix} 0.00000095 & 0 \\ 0 & 0.00000095 \end{pmatrix}$$

Initialization

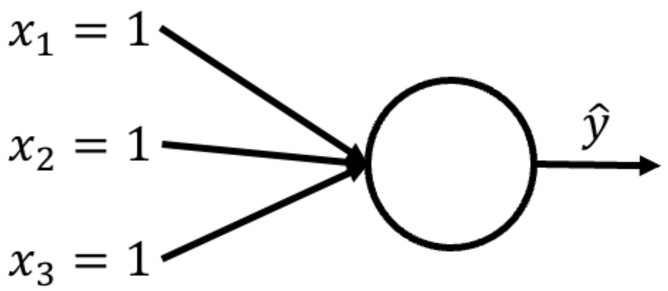


$$b = 0$$

$$z = w_1 x_1 + w_2 x_2$$

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2010.

Initialization

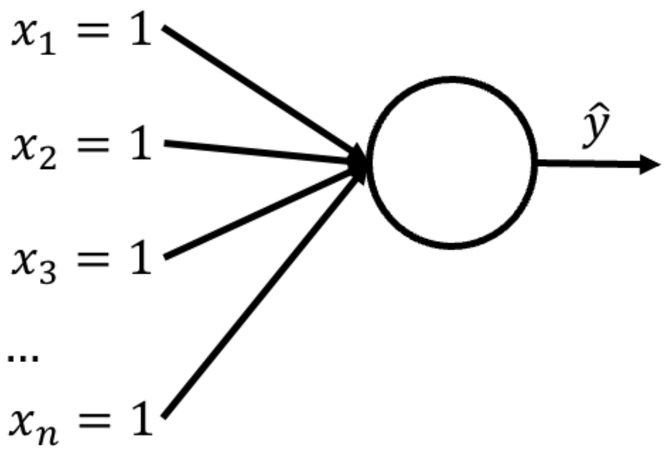


$$b = 0$$

$$z = w_1 x_1 + w_2 x_2 + w_3 x_3$$

Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings, 2010.

Initialization



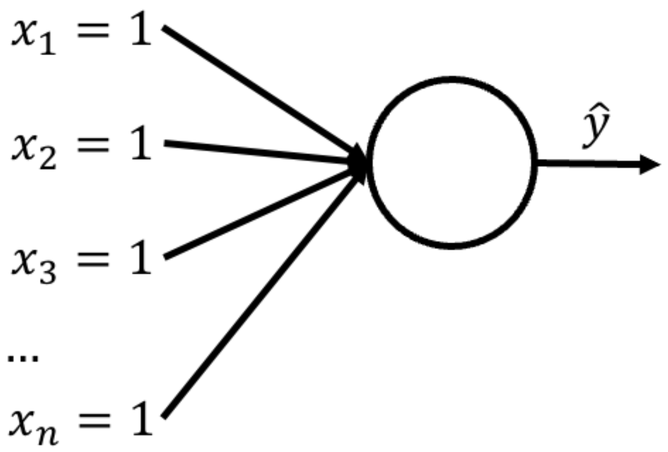
$$b = 0$$

$$Var(w_i) = 1$$

$$z = w_1x_1 + w_2x_2 + w_3x_3 + \ldots + w_nx_n$$

$$Var(z) = n\ Var(w_i)\ Var(x_i)$$

Initialization



$b = 0$

$$Var(w_i) = \frac{1}{n}$$

$$z = w_1 x_1 + w_2 x_2 + w_3 x_3 + \ldots + w_n x_n$$

$$Var(z) = n\ Var(w_i)\ Var(x_i) = n\ \frac{1}{n}\ Var(x_i)$$

Initialization

- Xavier Glorot Initialization

$$\sigma = \sqrt{\frac{2}{n_{in} + n_{out}}}$$

Sigmoid activation

- Kaiming He Initialization

$$\sigma = \sqrt{\frac{4}{n_{in} + n_{out}}}$$

ReLU activation

Normalization

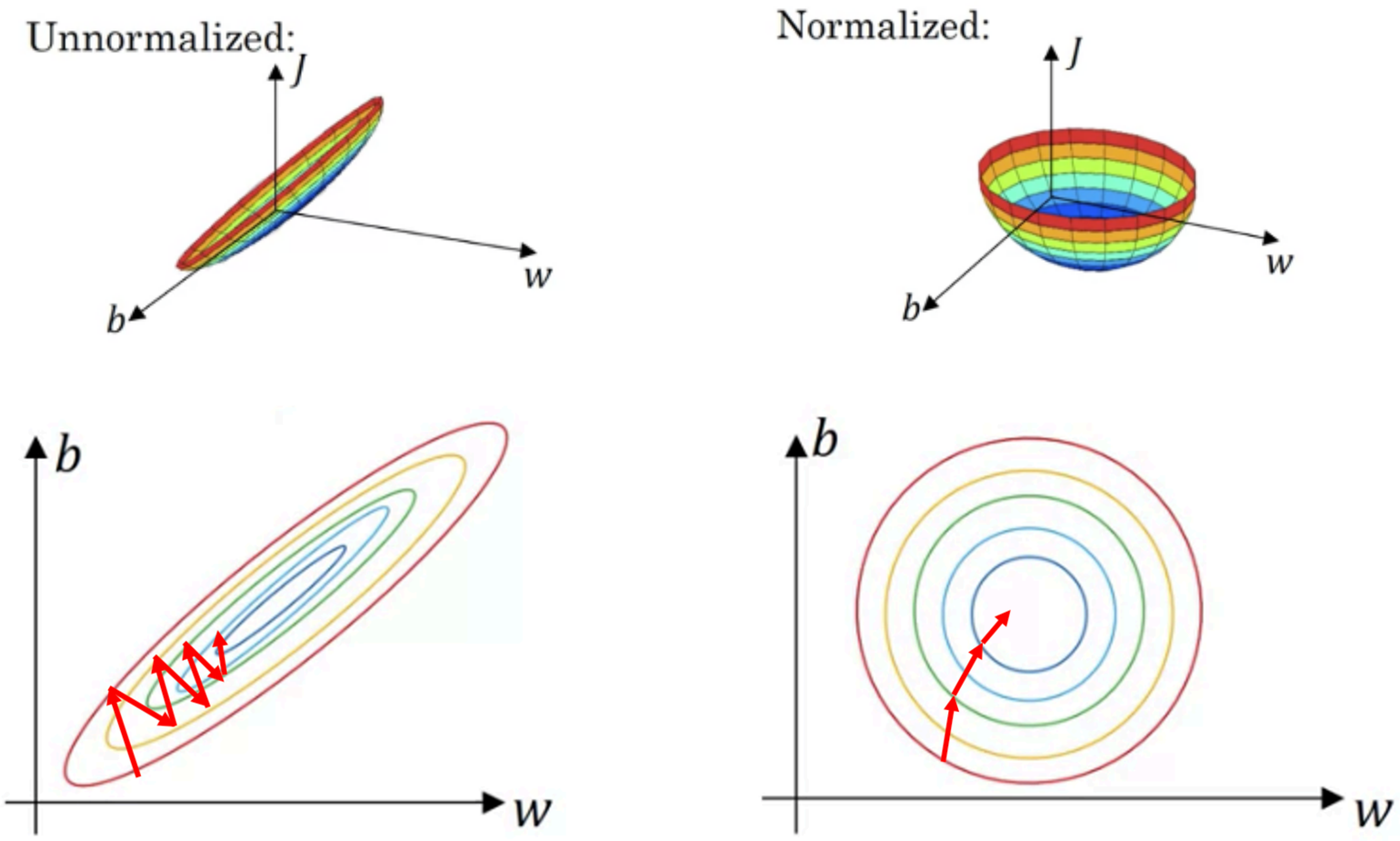


Image - Andrew Ng

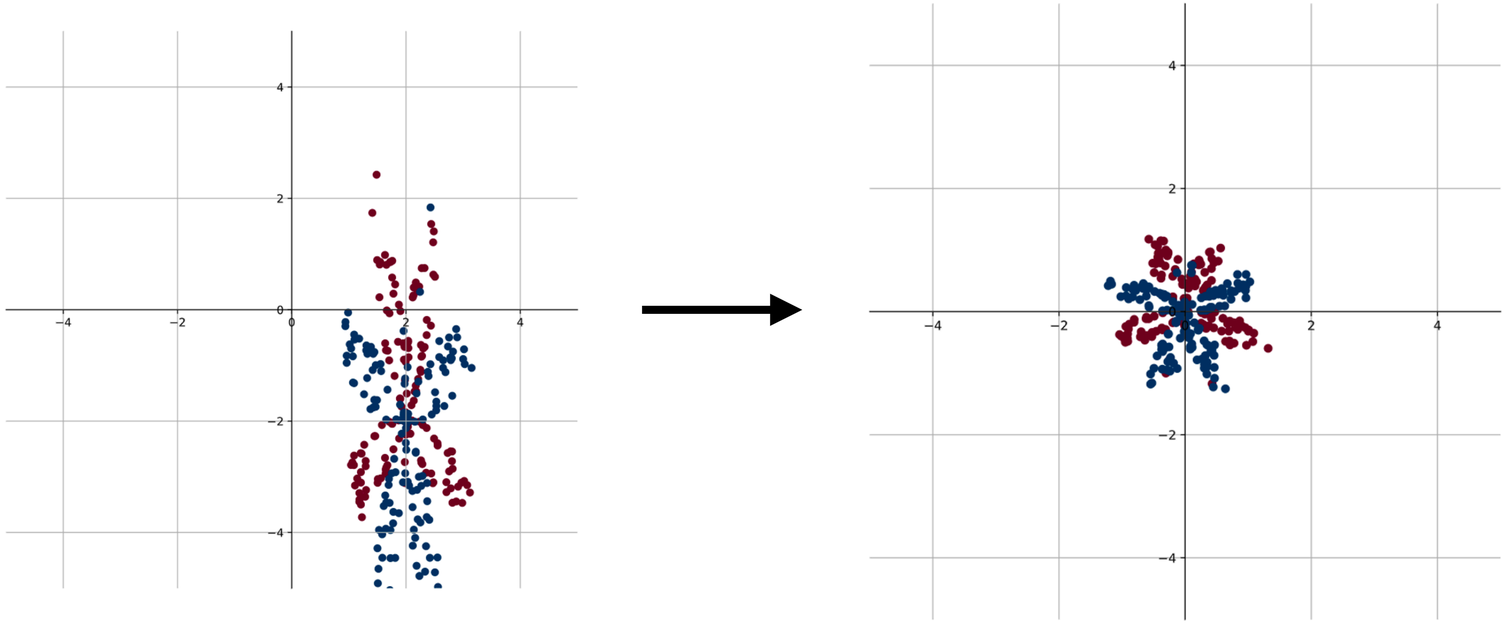
Normalization

- Mean
- Variance

$$\mu = E(X) = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

$$\sigma^2 = Var(X) = E\left[(X-\mu)^2\right]$$

$$\tilde{x} = \frac{x-\mu}{\sigma}$$



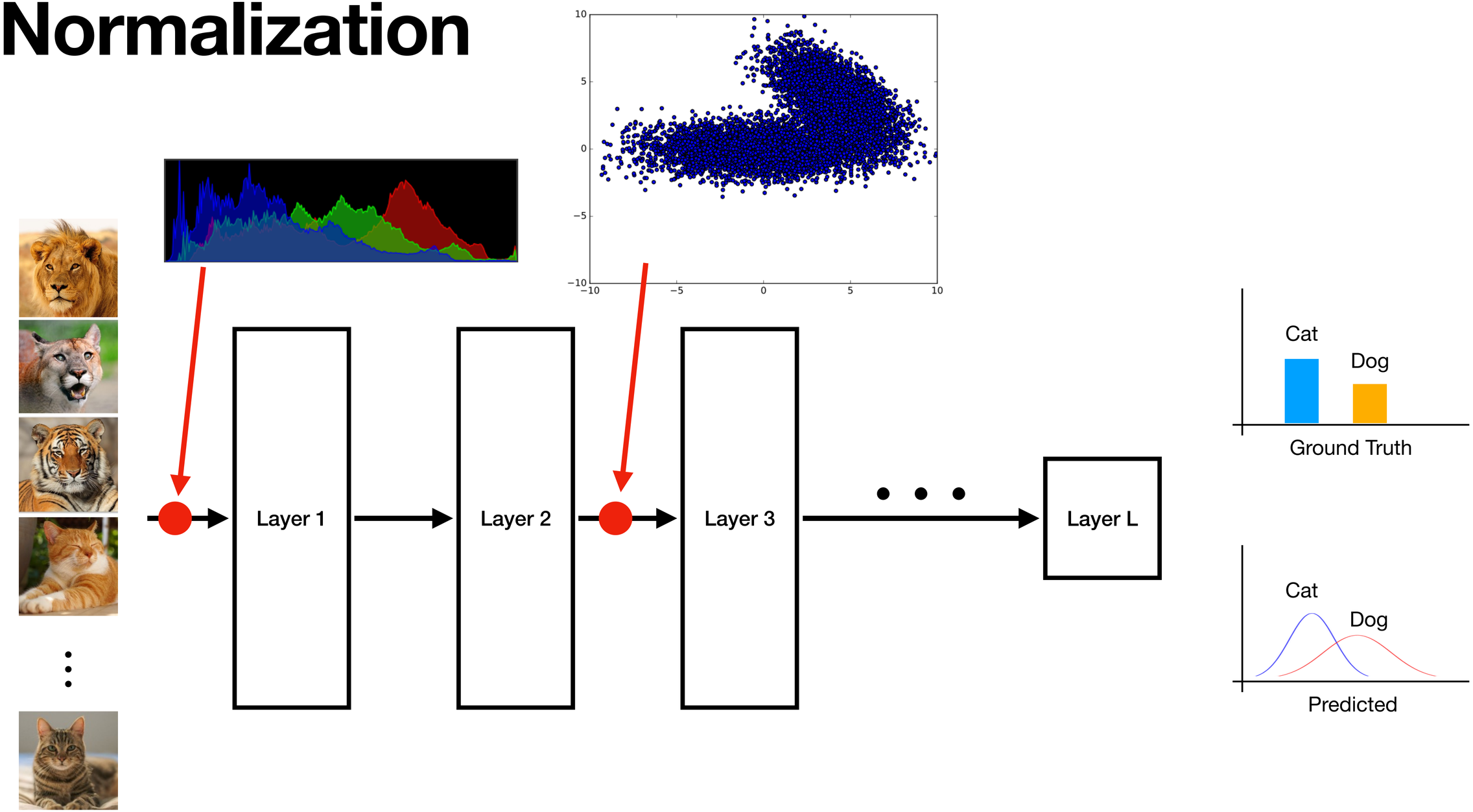
Normalization

- **Single sample** - has **value**

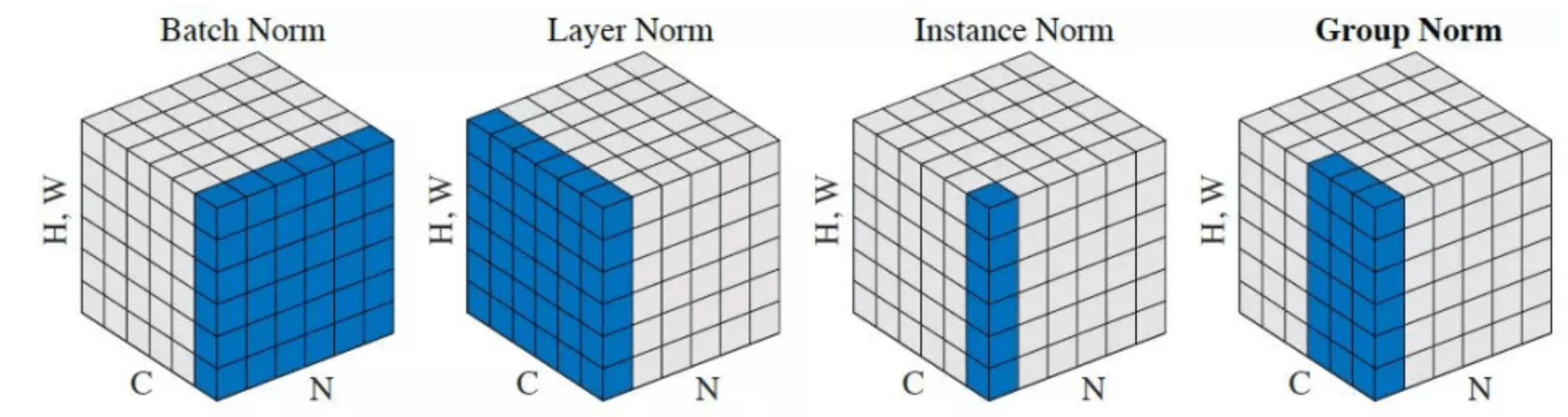
Normalization

- Single sample - has value
- **Set of samples** (Dataset) - can be thought of as **random variable**
    - Mean
    - Variance
    - Covariance
    - …

Normalization



Normalization



Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." *International conference on machine learning*. pmlr, 2015.

Normalization

Bude v písomke

- Normalization can also have learnable parameters

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}}\gamma + \beta$$

Small value
for numerical stability

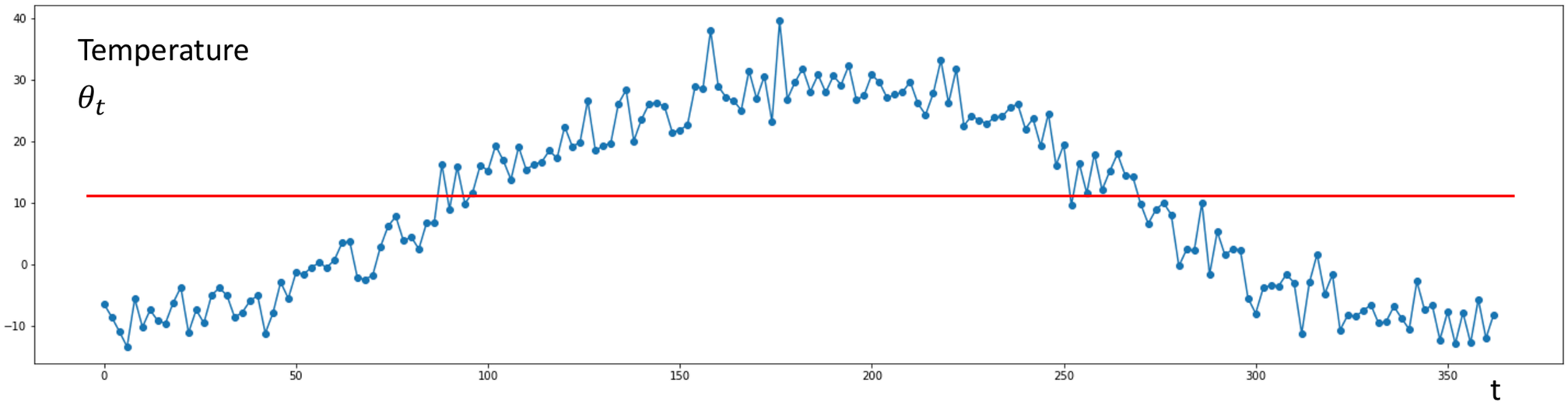
Normalization

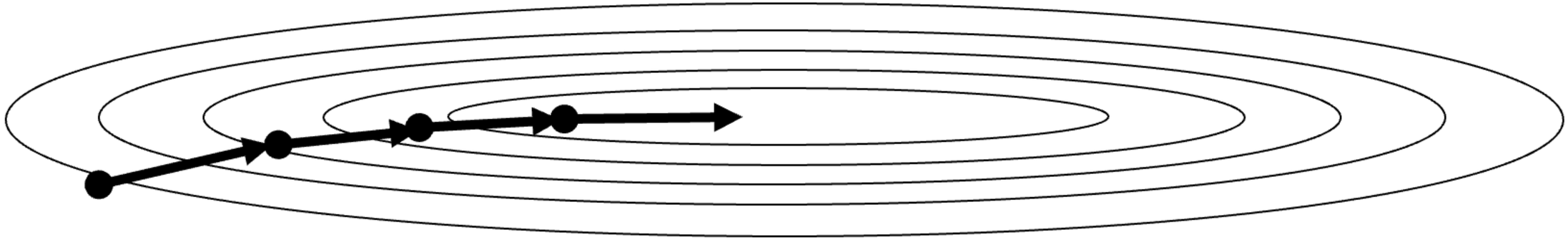
- Normalization can also have learnable parameters

- Training - we track the μ, σ values
- Validation - we must **use the same values** to transform data

4 Optimizers

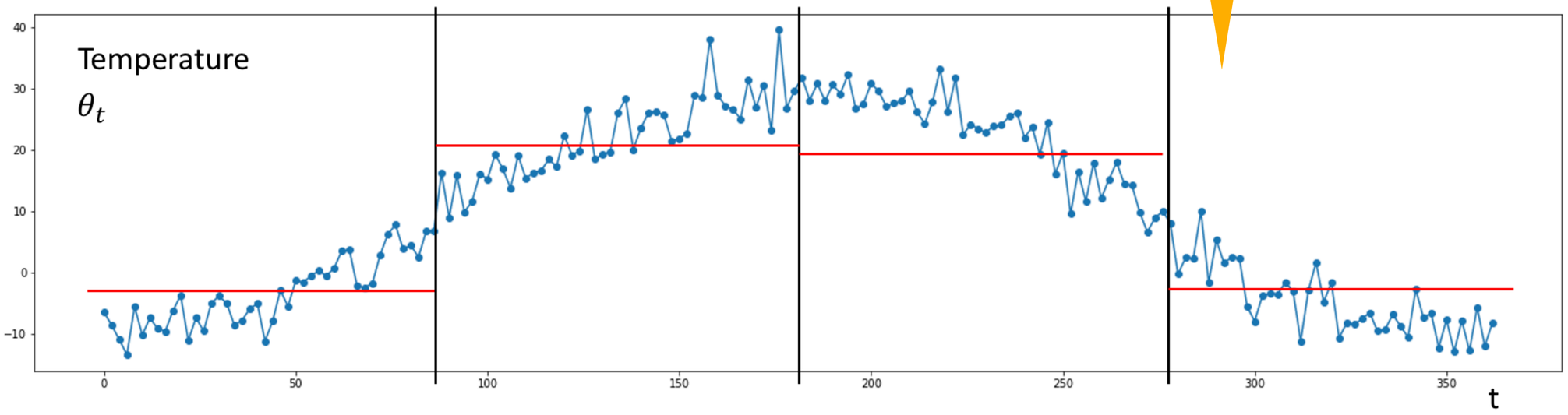
Optimizers - Example

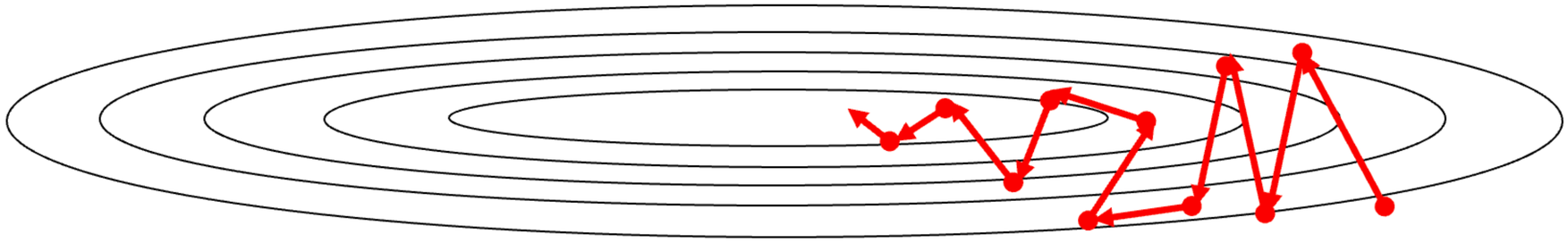




Optimizers - Example

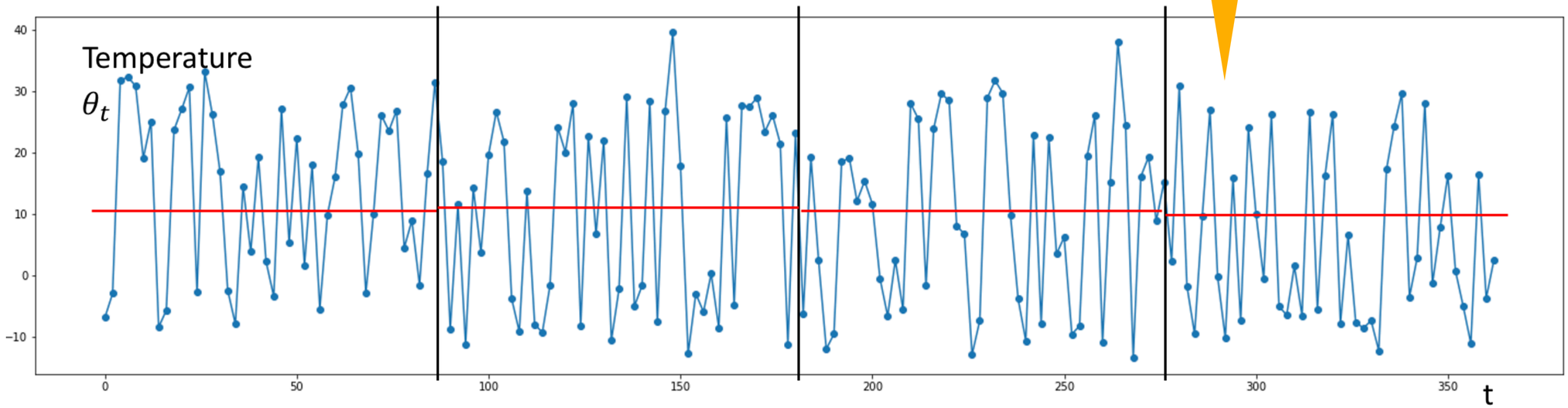
This is what happens if training mini-batches are not random

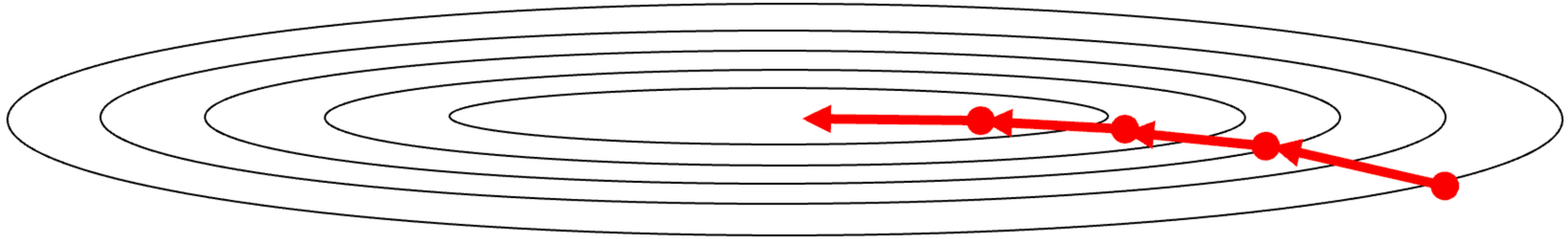




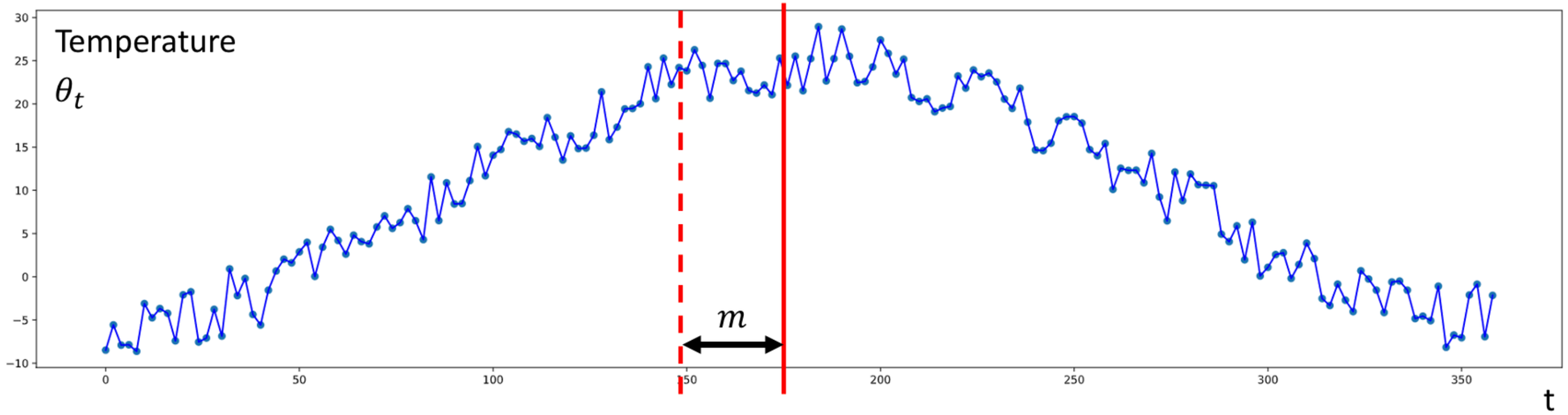
Optimizers - Example

Much better !



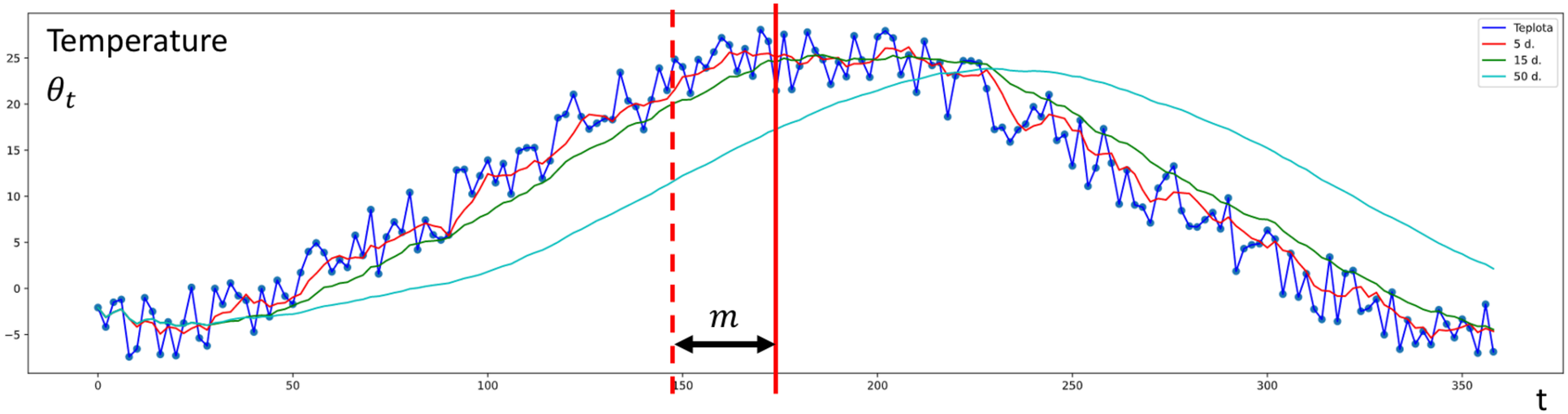


Average



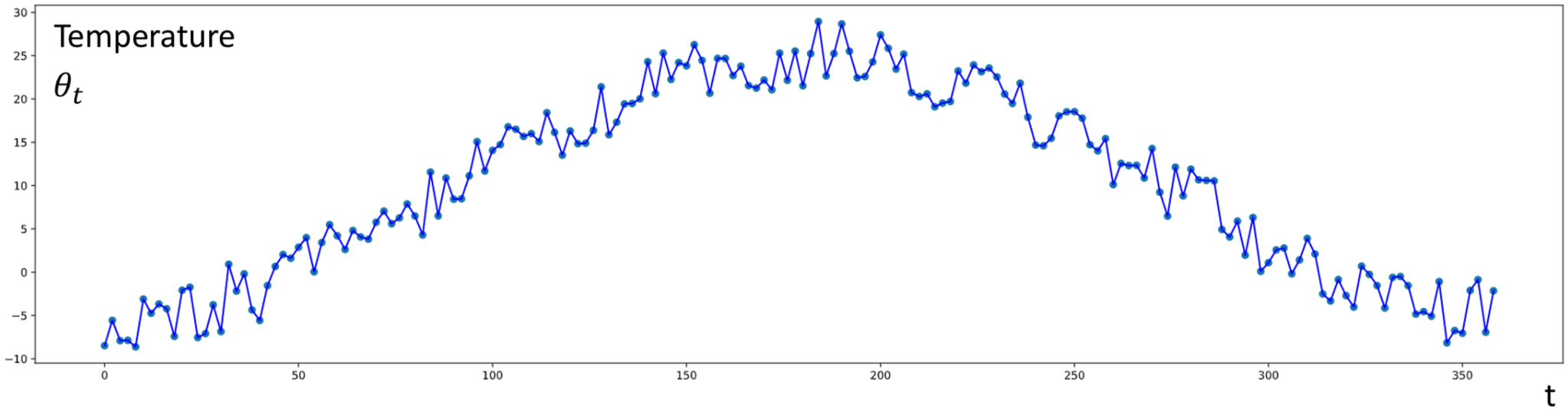
$$V_t = \frac{1}{m} \sum_{i=0}^{m-1} \theta_{t-i}$$

Average



$$V_t = \frac{1}{m} \sum_{i=0}^{m-1} \theta_{t-i}$$

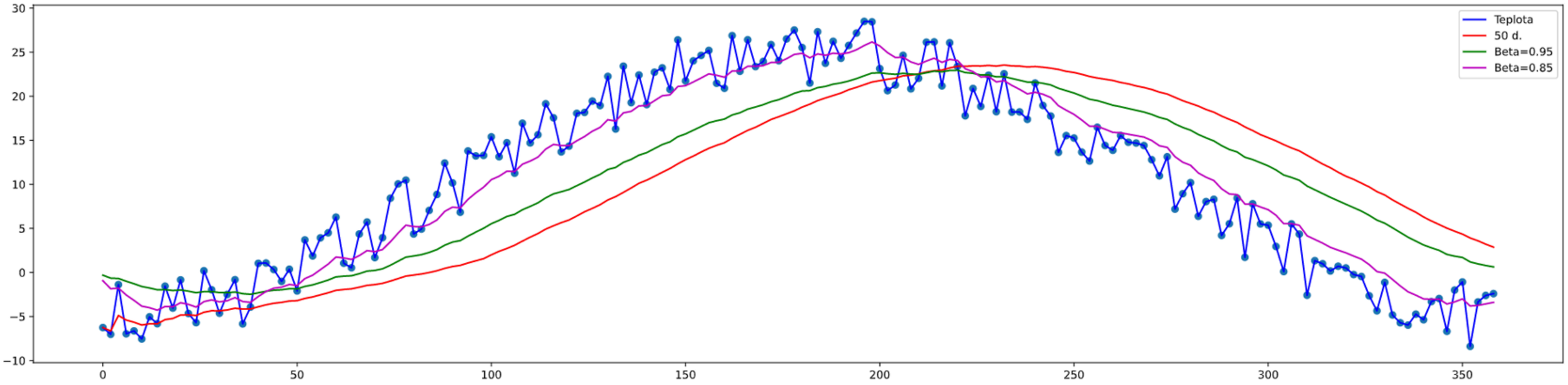
Exponentially Weighted Average



$$V_t = (1-\beta)\sum_{i=0}^{t}\beta^{(t-i)}\theta_i$$

$$V_t = (1-\beta)\theta_t + \beta V_{t-1}$$

Exponentially Weighted Average



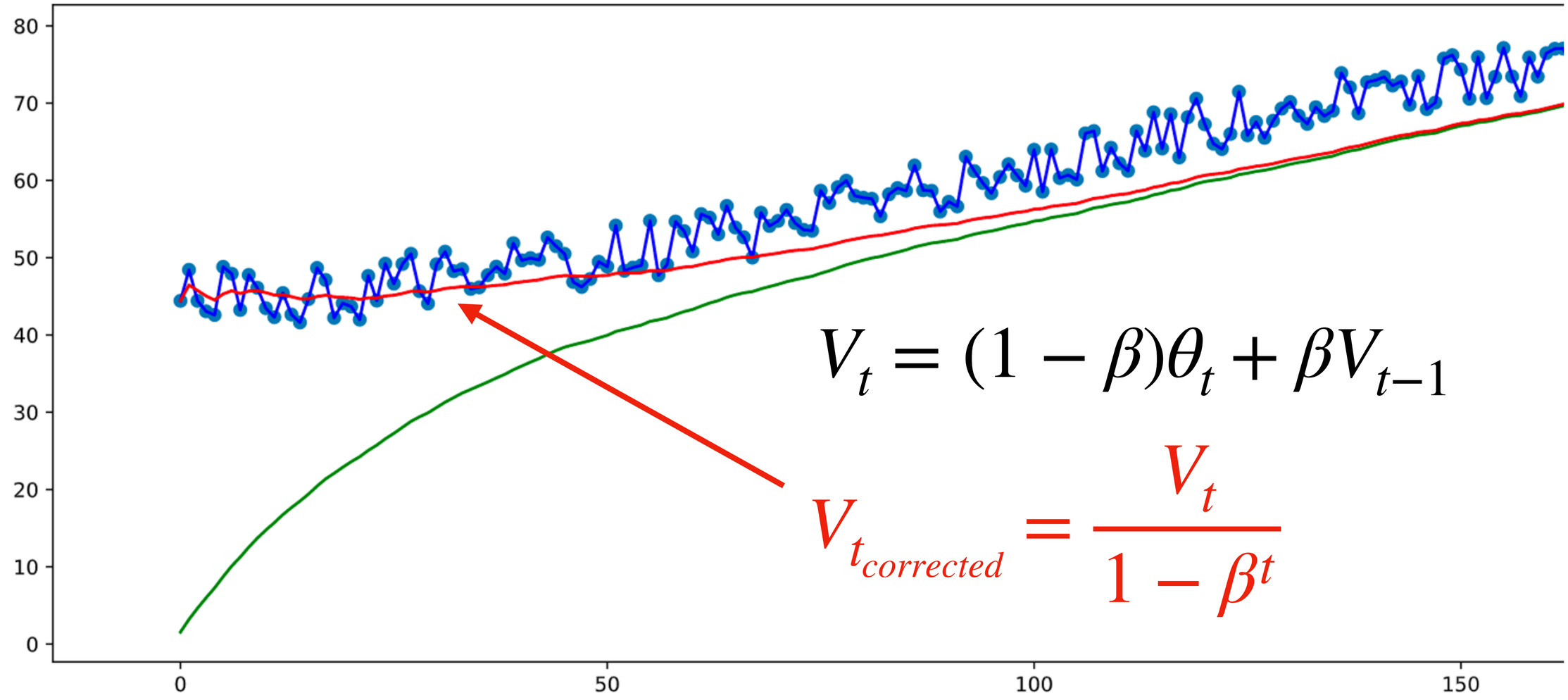
$$V_t = (1-\beta)\sum_{i=0}^{t}\beta^{(t-i)}\theta_i$$

$$V_t = (1-\beta)\theta_t + \beta V_{t-1}$$

Exponentially Weighted Average

Exponentially Weighted Average

Bias-Corrected Average



$$V_t = (1-\beta)\theta_t + \beta V_{t-1}$$

$$V_{t_{corrected}} = \frac{V_t}{1-\beta^t}$$

Problem With Gradient Descent

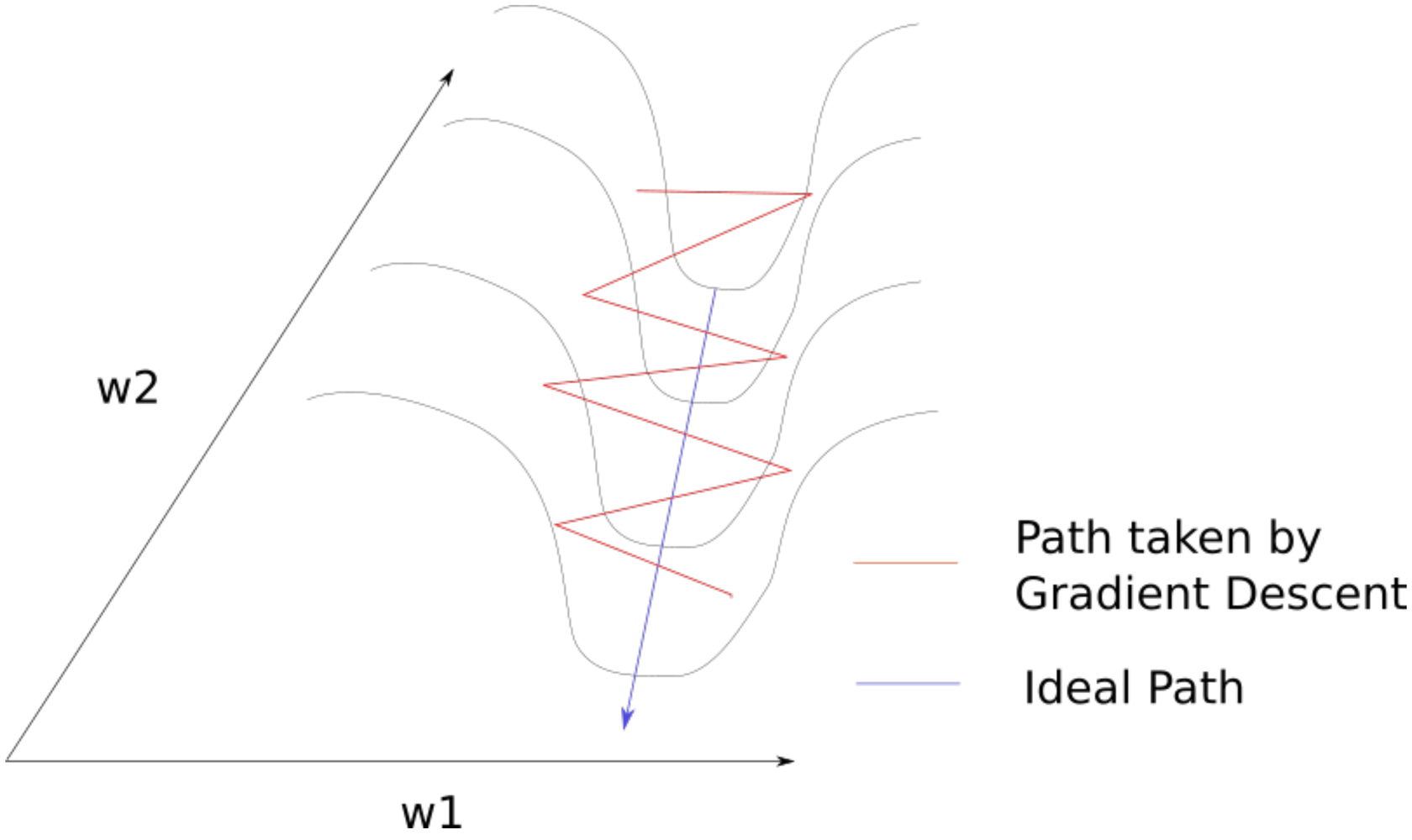
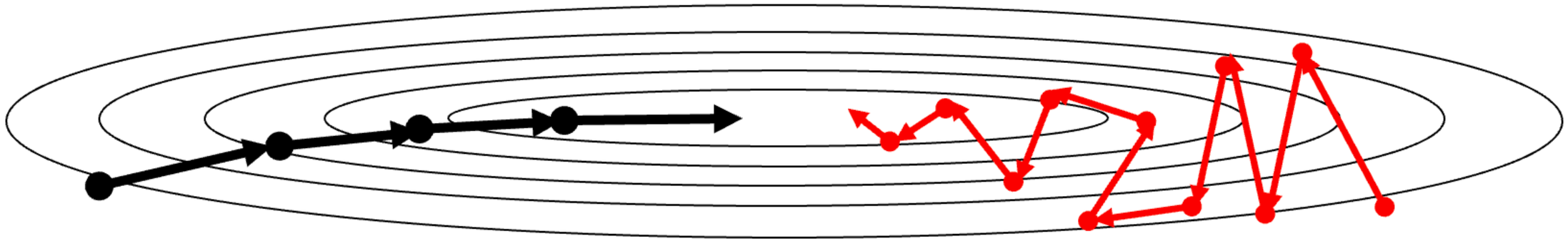


Image - https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/

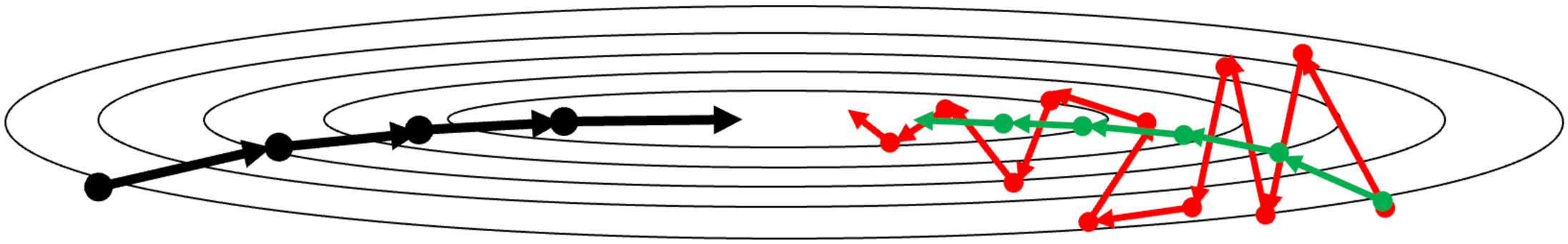
Gradient Descent - Momentum



Gradient Descent

$$W := W - \alpha dW$$
$$b := b - \alpha db$$

Gradient Descent - Momentum



Gradient Descent

$$W := W - \alpha dW$$
$$b := b - \alpha db$$

Gradient Descent with Momentum

$$V_{dW} := (1 - \beta)dW + \beta V_{dW}$$
$$V_{db} := (1 - \beta)db + \beta V_{db}$$

$$W := W - \alpha V_{dW}$$
$$b := b - \alpha V_{db}$$

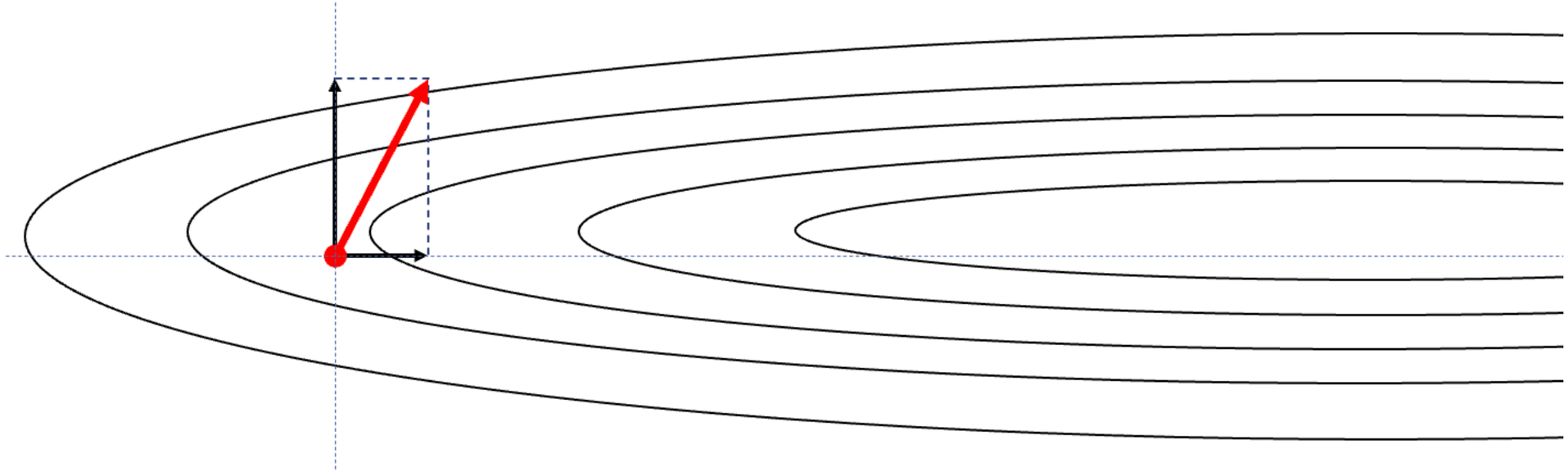
RMSProp

- Magnitude of gradients can differ for different weights
  - Can change during learning
  - Hard to choose a single global learning rate

- We keep a moving average of the squared gradient
  - Update weights by this magnitude

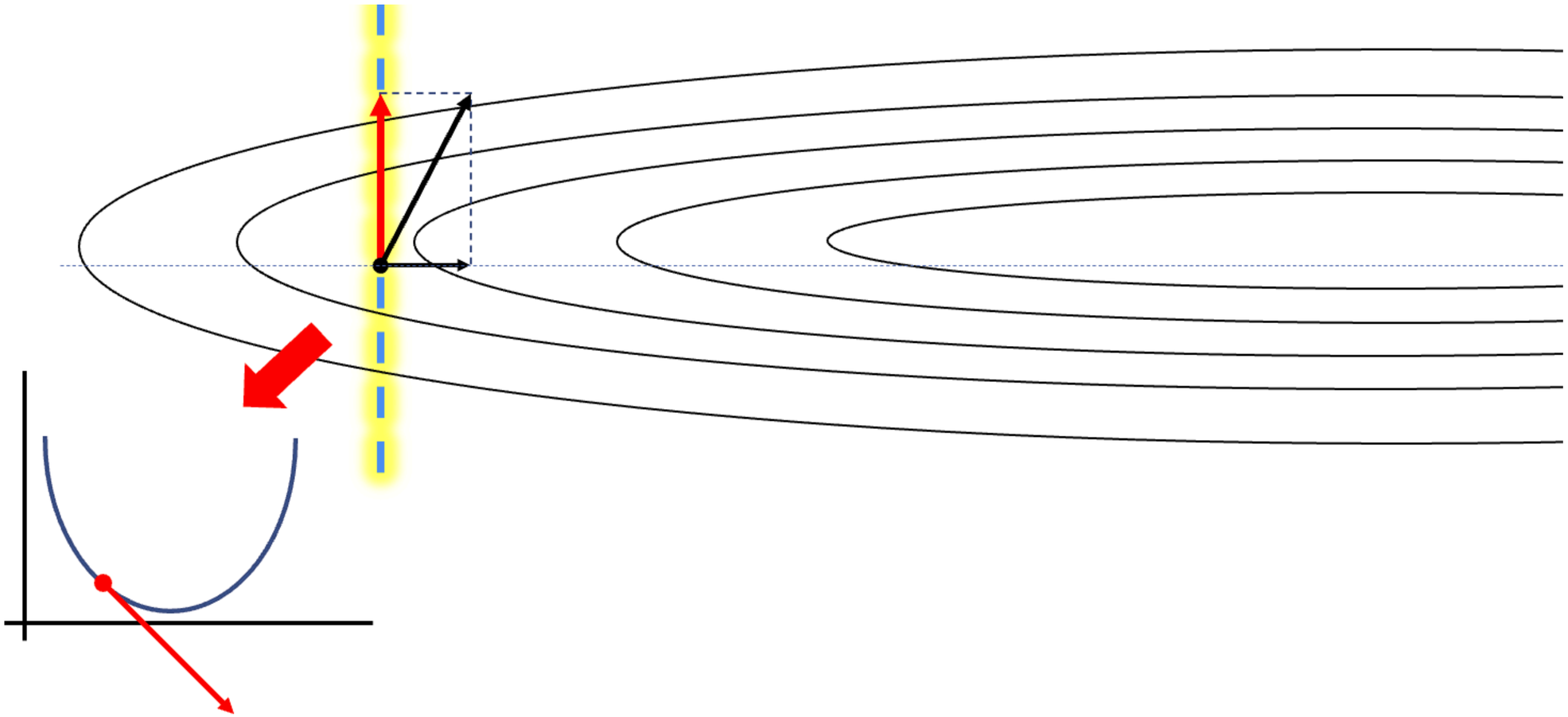
Unpublished, Proposed by Geoffrey Hinton

RMSProp



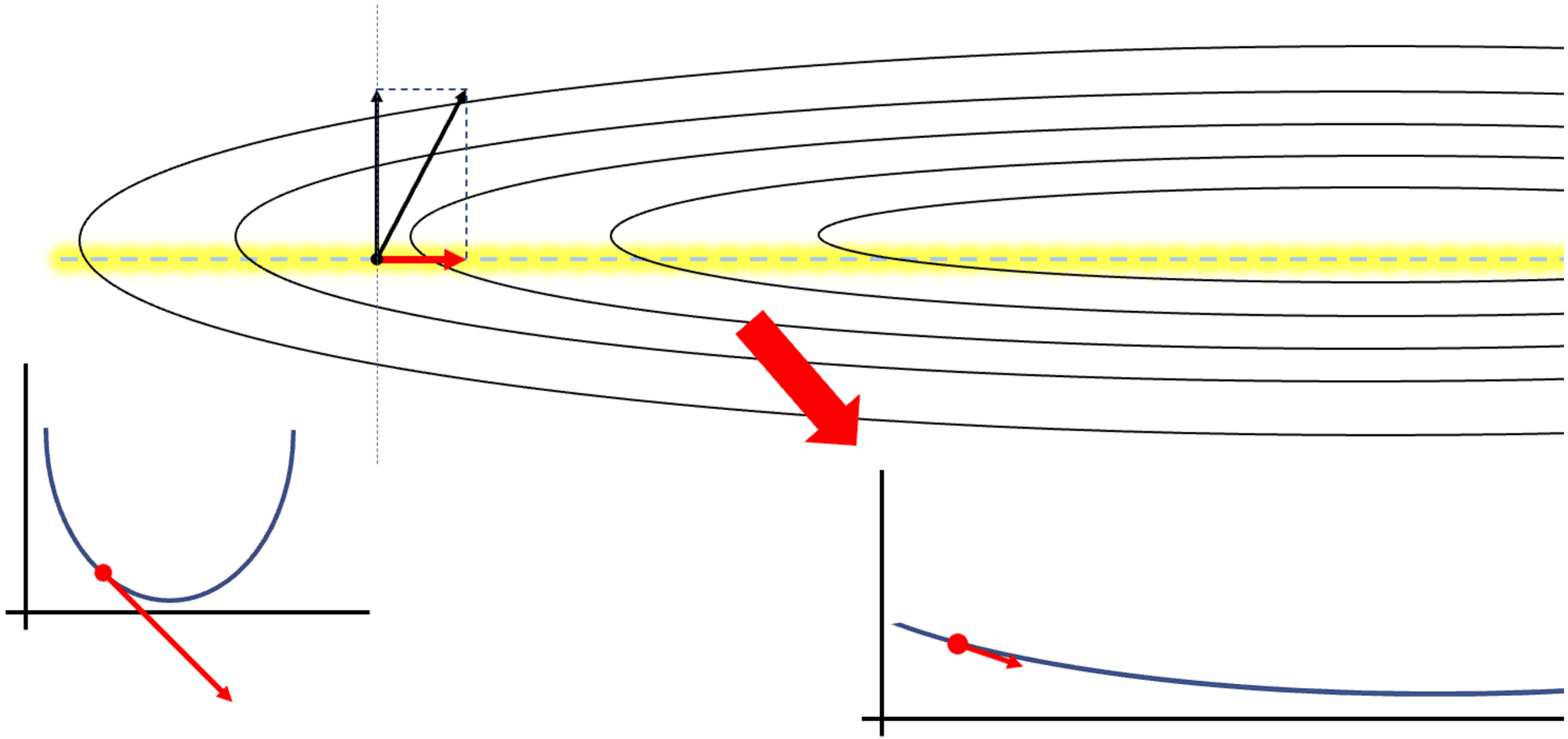
Unpublished, Proposed by Geoffrey Hinton

RMSProp



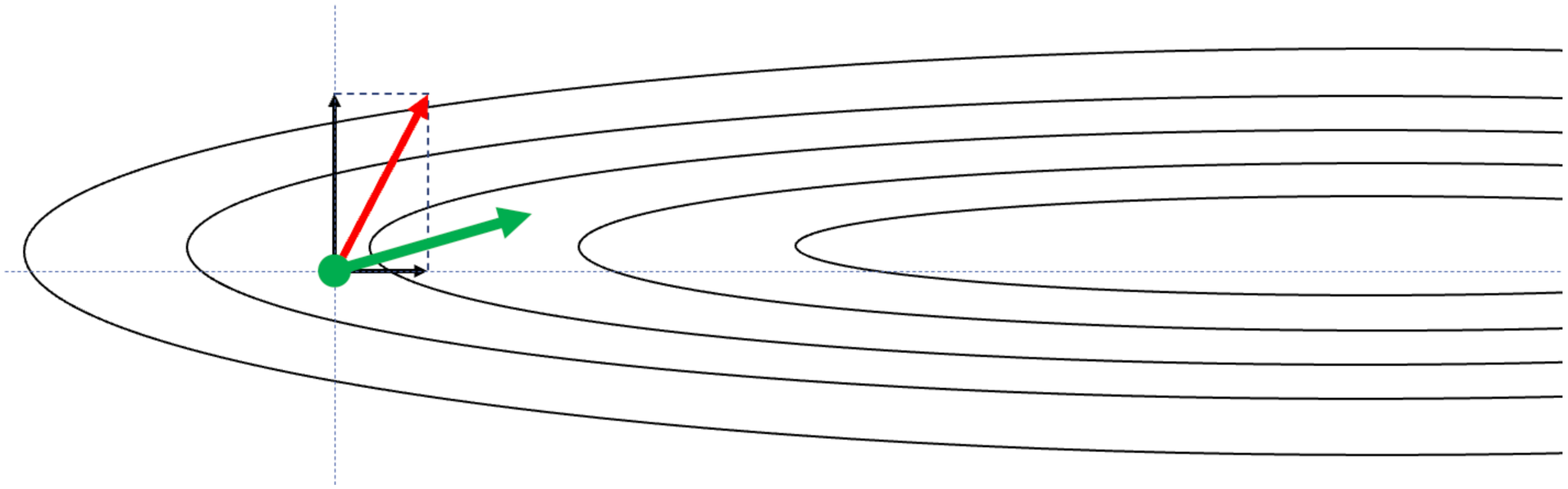
Unpublished, Proposed by Geoffrey Hinton

RMSProp



Unpublished, Proposed by Geoffrey Hinton

RMSProp



$$S_{dW} := (1-\beta)dW^2 + \beta S_{dW}$$

$$S_{db} := (1-\beta)db^2 + \beta S_{db}$$

$$W := W - \alpha \frac{dW}{\sqrt{S_{dW}}}$$

$$b := b - \alpha \frac{db}{\sqrt{S_{db}}}$$

Unpublished, Proposed by Geoffrey Hinton

Adam

- Combines ideas from momentum and RMSProp

Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980* (2014).

Adam

- Combines ideas from momentum and RMSProp

$$V_{dW} := (1 - \beta_1) dW + \beta_1 V_{dW}$$

First moment estimate

$$S_{dW} := (1 - \beta_2) dW^2 + \beta_2 S_{dW}$$

Second moment estimate

Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980* (2014).

Adam

- Combines ideas from momentum and RMSProp

$$V_{dW} := (1-\beta_1)dW + \beta_1 V_{dW}$$
$$S_{dW} := (1-\beta_2)dW^2 + \beta_2 S_{dW}$$

$$\widehat{V}_{dW} := \frac{V_{dW}}{1-\beta_1^t}$$

Bias-corrected first moment estimate

$$\widehat{S}_{dW} := \frac{S_{dW}}{1-\beta_2^t}$$

Bias-corrected second moment estimate

Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980* (2014).

Adam

- Combines ideas from momentum and RMSProp

$$V_{dW} := (1-\beta_1)dW + \beta_1 V_{dW}$$
$$S_{dW} := (1-\beta_2)dW^2 + \beta_2 S_{dW}$$

$$\widehat{V}_{dW} := \frac{V_{dW}}{1-\beta_1^t}$$
$$\widehat{S}_{dW} := \frac{S_{dW}}{1-\beta_2^t}$$

$$W := W - \alpha \frac{\widehat{V}_{dW}}{\sqrt{\widehat{S}_{dW}} + \epsilon}$$

Small value
for numerical stability

Kingma, Diederik P., and Jimmy Ba. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980* (2014).

Second-Order Methods

- First-order methods
  - gradient gives us **direction**
- Second-order methods
  - take curvature into account
  - Taylor series second—order approximation of function
  - estimate where the **minimum might be**

Newton's Method For Convex Functions

$$x := x - \alpha \nabla f(x)$$

Gradient Descent update rule

$$x := x - H(x)^{-1} \nabla f(x)$$

Inverse of Hessian matrix

Newton's Method update rule

Newton's Method For Convex Functions

- Hessian of $f(x_1, x_2, \ldots, x_n)$

$$\nabla^2 f(x_1, \ldots, x_n) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Second-Order Methods

- For large models (100M of parameters)
  - Hessian matrix can be **too big to be practical**
  - Inverting Hessian matrix can be too difficult to calculate

- Methods - Approximate Hessian
  - AdaHessian

Yao, Zhewei, et al. "Adahessian: An adaptive second order optimizer for machine learning." *proceedings of the AAAI conference on artificial intelligence*. Vol. 35. No. 12. 2021.