Artificial Intelligence and Machine Learning

Classification

Lecture 2: Outline

- Linear Regression (Review)
- Intro to Classification
- Logistic Regression (Linear Classifier)
- Classification Metrics
- Other ML Models
- Optimizers

Recap

$$\{(\mathbf{x}_i, y_i)\}_{i=1}^{N}, \mathbf{x}_i \in \mathbb{R}^n, y_i \in \mathbb{R}$$

Design your model

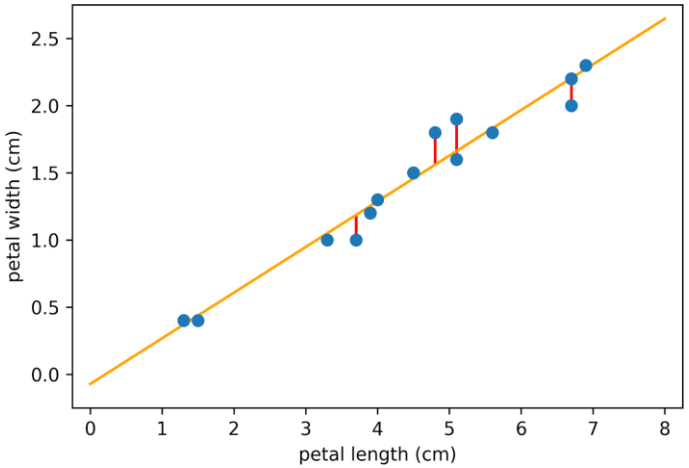
- Input scalar linear model (line fitting)
- Fitting polynomials (synthetically designing features from a one-dimensional input)

Design your loss function

- We used mean squared error loss throughout

Finding optimal parameter fitting

- Closed form solution to the linear least squares?
- Why is it linear least squares?
- Solution is closed form



Regression VS classification

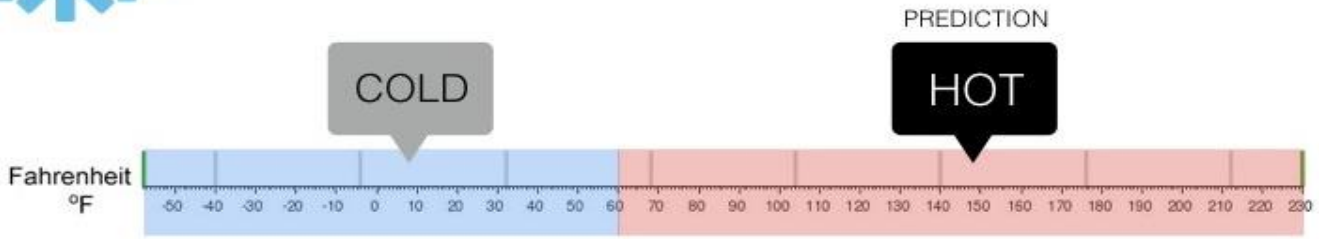
Regression

What is the temperature going to be tomorrow?

=> Continuous Values

Classification

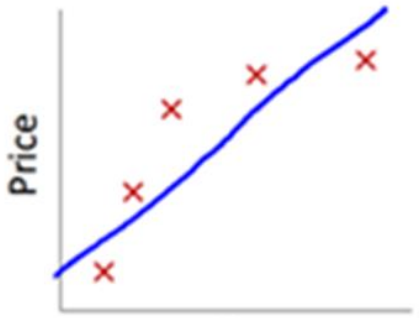
Will it be Cold or Hot tomorrow?



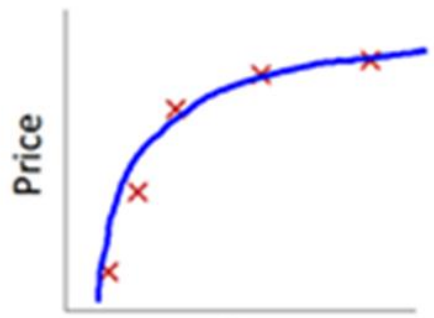
=> Discrete Values

Regression VS classification

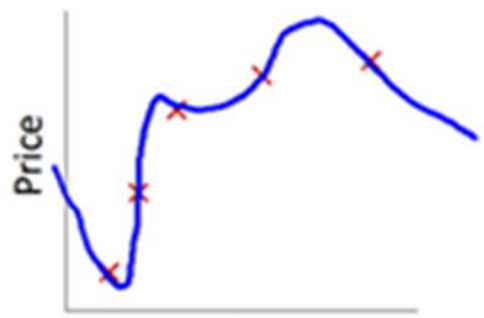
Regression:



Linear

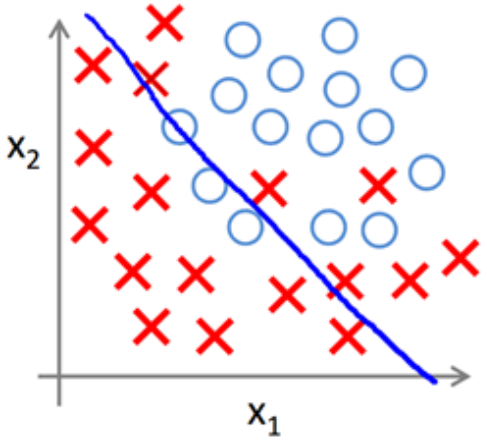


Polynomial

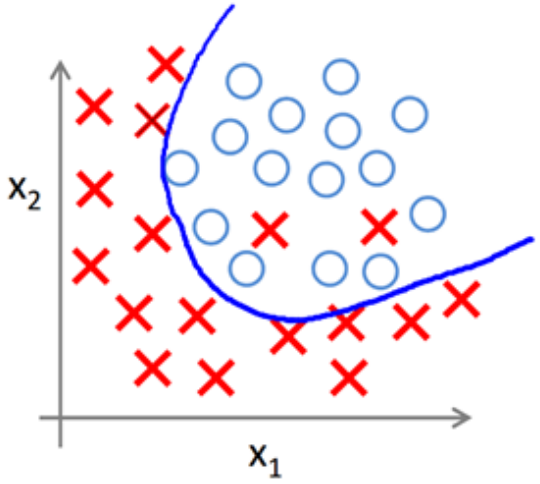


Very complex (overfitting)

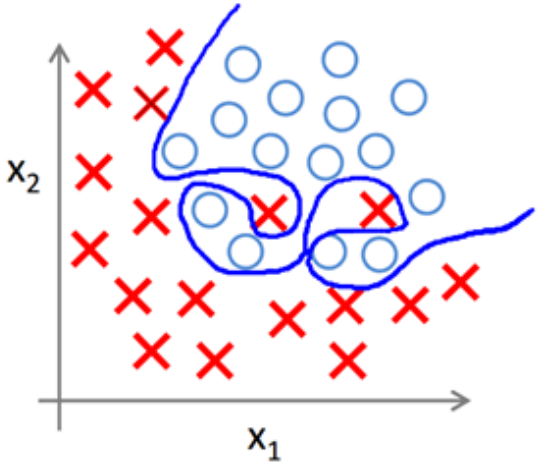
Classification:



Linear

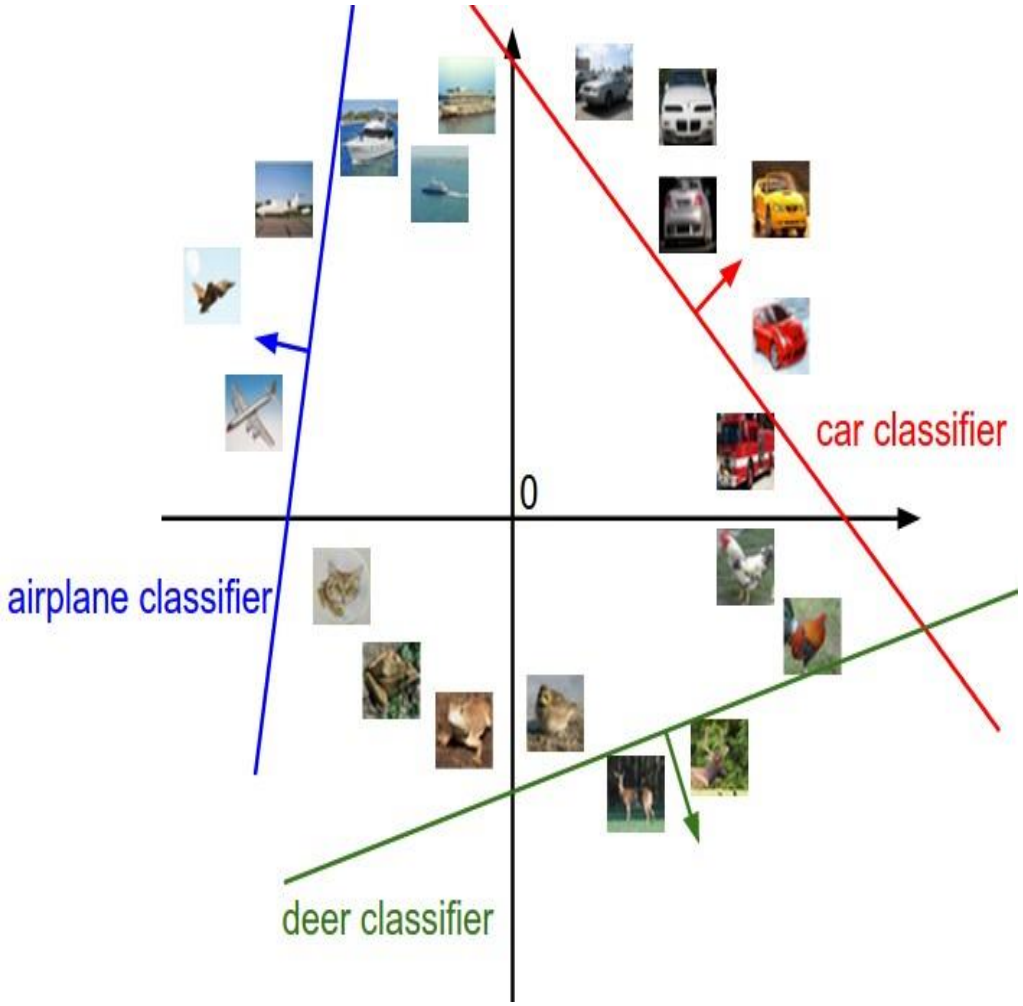


Polynomial



Very complex (overfitting)

Interpreting a Linear Classifier

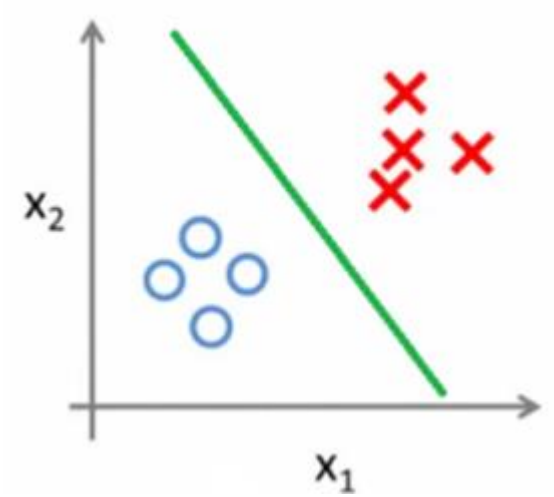


$$f(x_i, W, b) = Wx_i + b$$

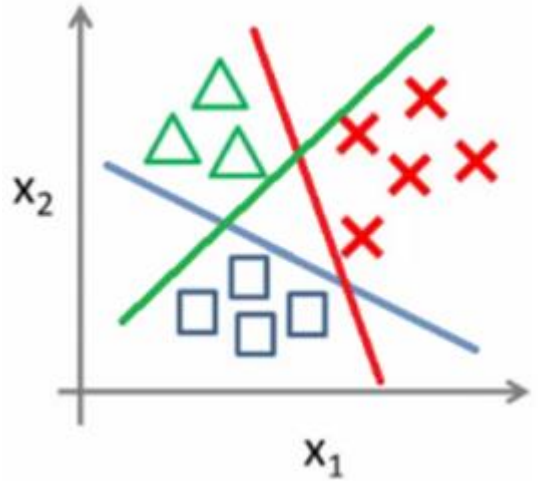
[32x32x3]
array of numbers 0...1
(3072 numbers total)

Classification Types

Binary Classification: Two possible outcomes (e.g., Yes/No, Spam/Not Spam).
How? Single Classifier.



Multiclass Classification: More than two outcomes (e.g., Class A, B, C).
How? Multiple *Single Classifiers*.

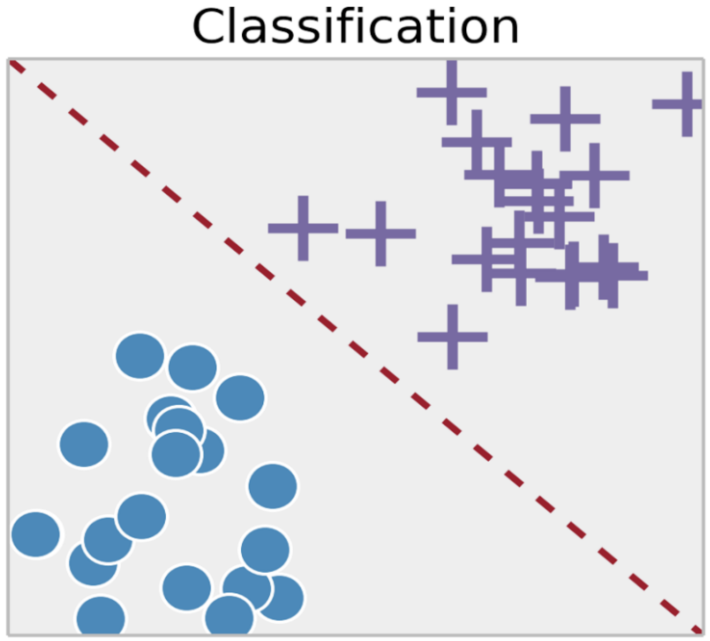


Linear Model in Disguise

$$\hat{y} = \mathbf{w}^T \mathbf{x}$$

$$\mathbf{w} = [w_0, w_1, \cdots, w_m]^T$$

$$\mathbf{x} = [1, x^1, \cdots, x^m]^T$$



$$\{(\mathbf{x}_i, y_i)\}_{i=1}^N, \mathbf{x}_i \in \mathbb{R}^n, y_i \in \{0, 1\}$$

Linear Model in Disguise

$$\hat{y} = \mathbf{w}^T \mathbf{x}$$

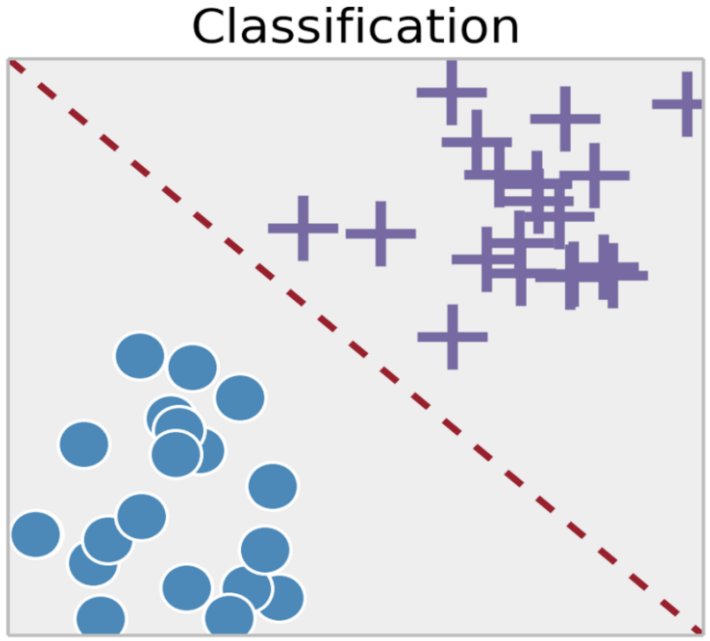
$$\mathbf{w} = [w_0, w_1, \cdots, w_m]^T$$

$$\mathbf{x} = [1, x^1, \cdots, x^m]^T$$

$$\hat{y} \approx y$$

Recall that the output/label y isbinary

How to map the predictions to binary?



$$\{(\mathbf{x}_i, y_i)\}_{i=1}^N, \mathbf{x}_i \in \mathbb{R}^n, y_i \in \{0, 1\}$$

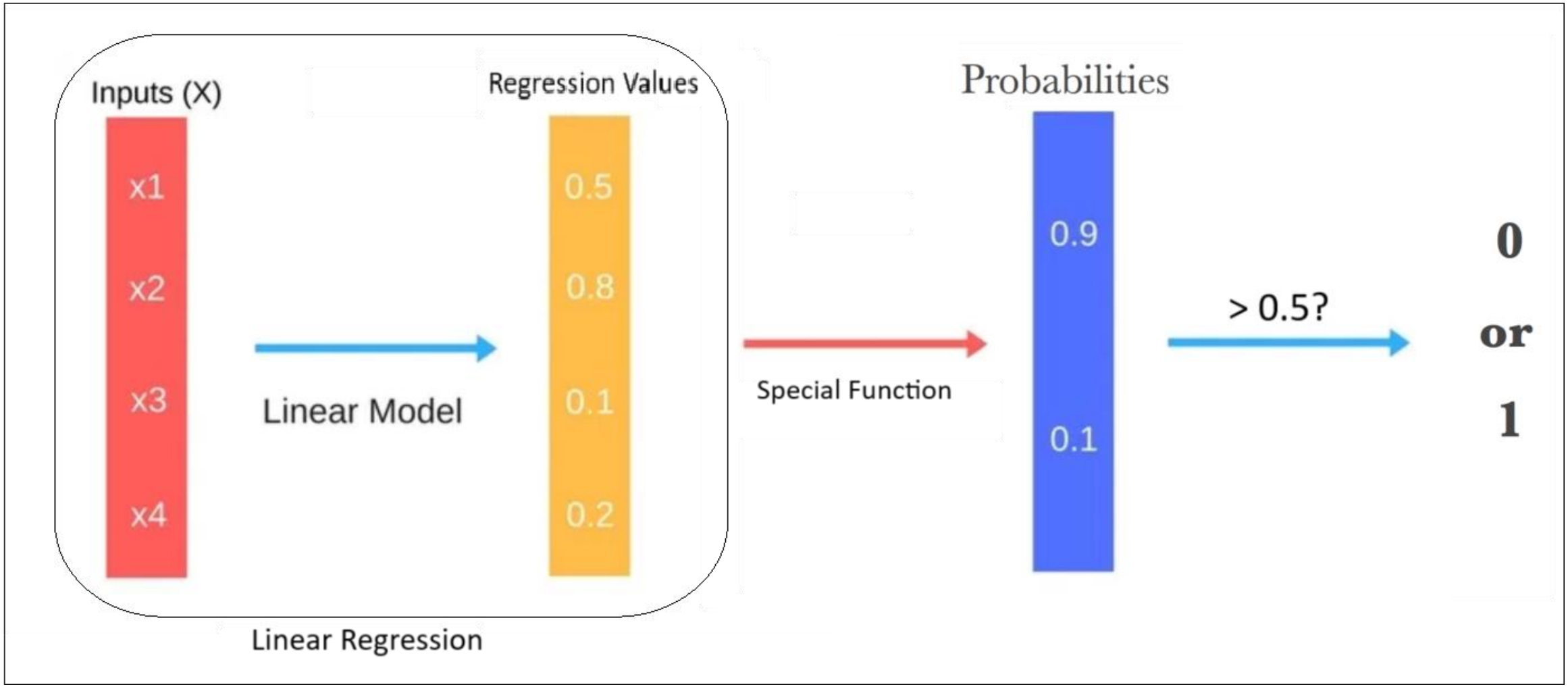
Linear Classifier

The linear classifier we will study is basically a **regression** model with two key modifications:

- Pass the regression outputs through a **special function** to convert them into probabilities.
- Replace the Mean Squared Error loss with a **new loss function** designed for probabilities.

Applying these two to the Linear Regression should give Linear Classifier (called **Logistic Regression**).

Logistic Regression



Special Function

We are searching for a function that has the following characteristics:

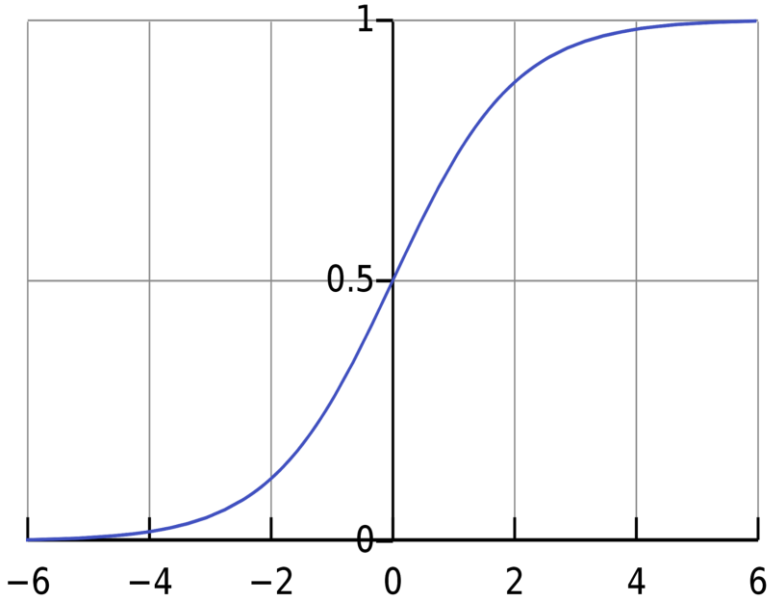
1. Could convert any arbitrary input values to **[0,1] range (Probabilities)**.
2. **Smooth and Differentiable**. Because we want to find the minima later, right?

Any ideas?

Sigmoid Function (for Binary Classification)

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

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



Assuming you have negative and positive classes, its output would be the probability of the positive class.

But what if we have multiclass problem?🤔

$$\lim_{z \to -\infty} \sigma(z) = 0$$

$$\lim_{z \to \infty} \sigma(z) = 1$$

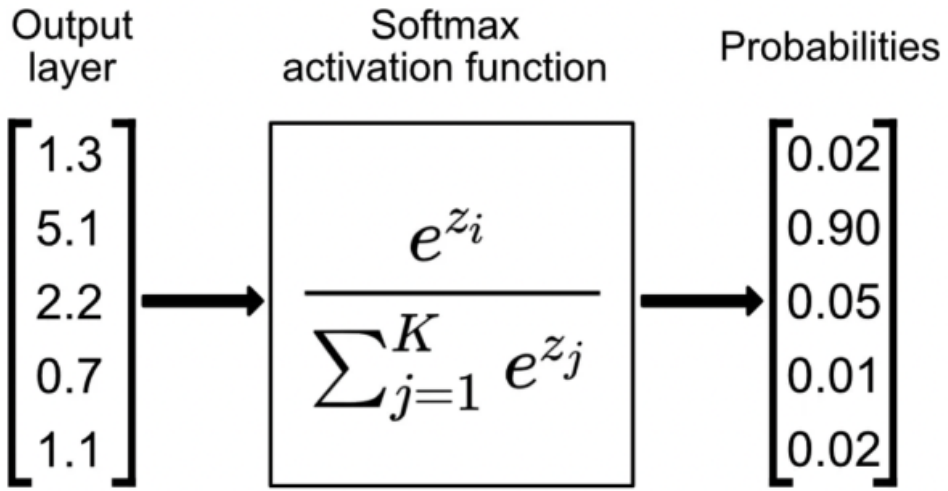
Softmax Function (for Multiclass Classification)

$$Softmax(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{n} \exp(z_j)}$$

Numerator: The exponential of the ith class $\exp(z_i)$.
Denominator: The sum of the exponentials of all classes $\sum_{j=1}^{n} \exp(z_j)$.

E.g. Divide the cat value by the cat, dog and deer values to get the cat probability.

Assuming you have K classes, its output would be **the probability of the ith class**. So, you will run it K times to get the probability of each class.



Note: Sigmoid is a special case of Softmax.
Optional: Can you prove it?

Classification Loss

We are searching for a loss that have the following characteristics:

1. **Probabilistic behaviour**: Should work with probabilities, not raw numbers.
2. **Sensitivity**: It should heavily penalize incorrect confident predictions (e.g., predicting 0.9 when the true label is 0).
3. **Differentiable**: Must be smooth and differentiable to enable optimization.

Any ideas?

Classification Loss

We are searching for a loss that have the following characteristics:

1. **Probabilistic behaviour**: Should work with probabilities, not raw numbers.
2. **Sensitivity**: It should heavily penalize incorrect confident predictions (e.g., predicting 0.9 when the true label is 0).
3. **Differentiable**: Must be smooth and differentiable to enable optimization.

Any ideas?

Logarithm is all you need :)

Binary Cross Entropy (LogLoss)

- For one sample, this can be represented mathematically by:

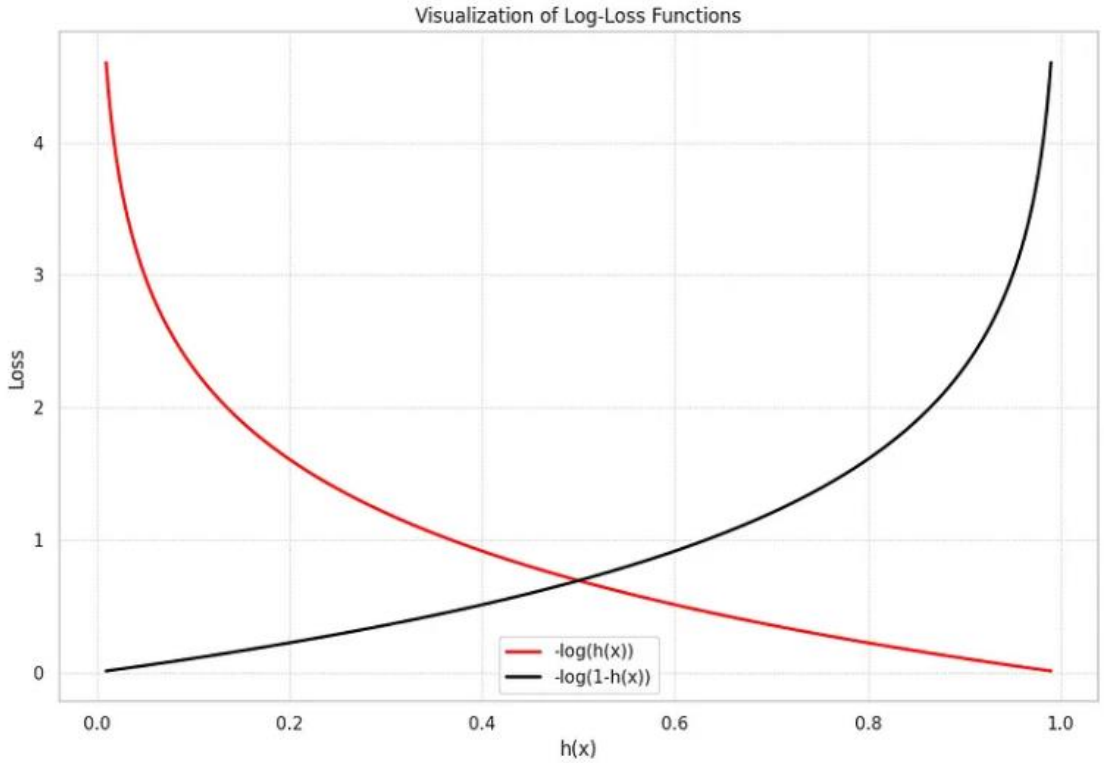
$$loss(y,p) = \begin{cases} -\log(p), & if\ y = 1 \\ -\log(1-p), & if\ y = 0 \end{cases}$$

Where y is the label, and p is the predicted probability.

If $y = 1$ and prediction is close to 1, loss will be close to **zero**. (Great!)

If $y = 1$ and prediction is close to 0, loss will be close to **infinity**. (Very bad!)

Same behavior for $y = 0$



Binary Cross Entropy (LogLoss)

- For one sample, this can be represented mathematically by:

$$loss(y,p) = \begin{cases} -\log(p), & if\ y = 1 \\ -\log(1-p), & if\ y = 0 \end{cases}$$

- Let's rewrite it in one line:

$$loss(y,p) = -(y * \log(p) + (1-y) * \log(1-p))$$

But we have many samples, not only one, right?

Binary Cross Entropy (LogLoss)

- For one sample, this can be represented mathematically by:

$$loss(y,p) = \begin{cases} -\log(p), & if\ y = 1 \\ -\log(1-p), & if\ y = 0 \end{cases}$$

- Let's write it in one line:

$$loss(y,p) = -(y * \log(p) + (1-y) * \log(1-p))$$

- Sum and average:

$$logloss(Y,P) = -\frac{1}{N}\sum_{i=1}^{N}(y_i * \log(p_i) + (1-y_i) * \log(1-p_i))$$

Y: Represents the true labels.

P: Represents the predicted probabilities for the positive class.

How to find optimal Parameters?

$$J(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\sigma(\mathbf{w}^T\mathbf{x}_i)) + (1-y_i)\log(1-\sigma(\mathbf{w}^T\mathbf{x}_i))$$

Just like before, simply take

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = 0$$

However, this does not have a nice closed solution,
unlike the MSE case

How to find optimal Parameters?

$$J(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\sigma(\mathbf{w}^T\mathbf{x}_i)) + (1-y_i)\log(1-\sigma(\mathbf{w}^T\mathbf{x}_i))$$

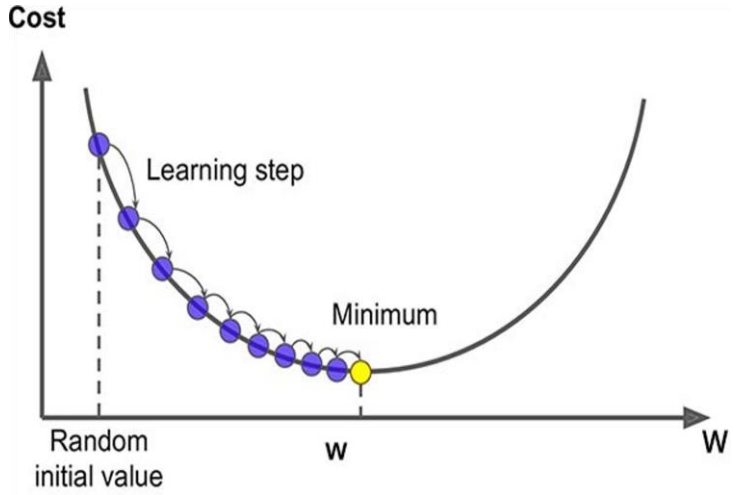
Just like before, simply take

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = 0$$

However, this does not have a nice closed solution unlike the MSE case

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \eta \nabla_{\mathbf{w}} J(\mathbf{w}^k)$$

Learning rate



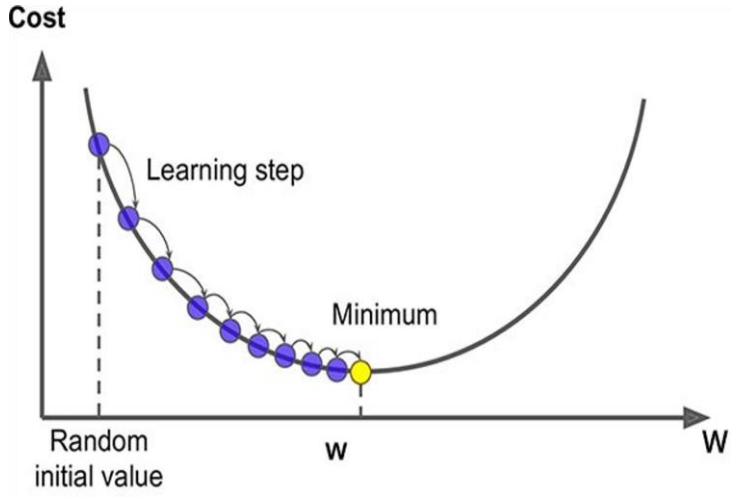
Instead, use gradient descent (optimizer)!

Logistic Regression

$$J(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\sigma(\mathbf{w}^T\mathbf{x}_i)) + (1-y_i)\log(1-\sigma(\mathbf{w}^T\mathbf{x}_i))$$

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \eta\nabla_{\mathbf{w}} J(\mathbf{w}^k)$$

Learning rate



- We have a linear model for prediction
- For classification, we want to output a probability
- We map the prediction to probabilities with a sigmoid function
- We have a loss function (BCE) to compare models

Logistic Regression

Linear Regression	Logistic Regression
For Regression	For Classification
We predict the target value for any input value	We predict the probability that the input value belongs to the specific target
Target: Real Values	Target: Discrete values
Graph: Straight Line	Graph: S-curve

Classification Metrics

- **Accuracy:**

  Accuracy measures the proportion of correctly classified samples out of the total number of samples.

$$Accuracy = \frac{Correct\ Samples}{All\ Sample}$$

Question: Why not optimizing for the accuracy directly instead of using LogLoss?

Classification Metrics

- **Accuracy:**

  Accuracy measures the proportion of correctly classified samples out of the total number of samples.

$$Accuracy = \frac{Correct\ Samples}{All\ Samples}$$

Question: Why not optimizing for the accuracy directly instead of using LogLoss?

Non-differentiable

Classification Metrics

- **Accuracy:**

  Accuracy measures the proportion of correctly classified samples out of the total number of samples.

$$Accuracy = \frac{Correct\ Samples}{All\ Sample}$$

Could also be written as:

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

Where:

TP: True Positives

TN: True Negatives

FP: False Positives

FN: False Negatives

Question: Why not optimizing for the accuracy directly instead of using LogLoss?
Non-differentiable

Classification Metrics

But accuracy isn’t always enough!

Imagine an imbalanced dataset with 95% negative labels (e.g., "not spam").

A model predicting "negative" all the time will be 95% accurate but useless.

Classification Metrics

But accuracy isn't always enough!

Imagine an imbalanced dataset with 95% negative labels (e.g., "not spam").
A model predicting "negative" all the time will be 95% accurate but useless.

Different goals require different metrics

- Do you care more about **catching all positives** (e.g., cancer detection)?
- Or avoiding **false alarms** (e.g., fraud detection)?

Classification Metrics

- **Precision:**

  Among all the positive predictions the model made,
  how many are actually correct?
  important when **minimizing false alarms** is critical,
  like in fraud detection or spam filtering.

$$Precision = \frac{True\ Positives\ (TP)}{True\ Positives\ (TP) + False\ Positives\ (FP)}$$

Classification Metrics

- **Recall:**

  Among all the actual positives, how many did the model correctly identify?

  important when **catching all positives** is vital, such as in cancer detection.

$$Recall = \frac{True\ Positives\ (TP)}{True\ Positives\ (TP) + False\ Negatives\ (FN)}$$

Classification Metrics

- **F1 Score:**

  What if both precision and recall are important?

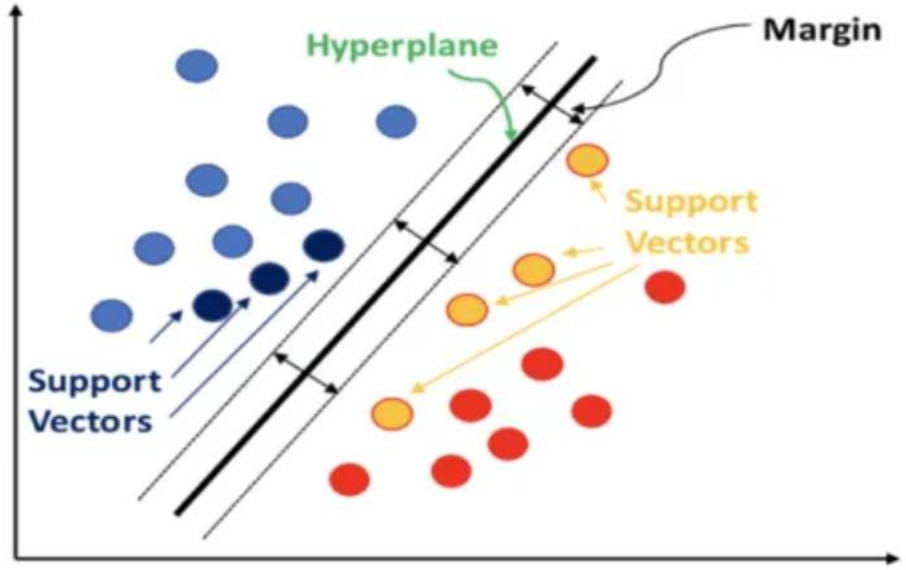
  Take their (harmonic) mean.

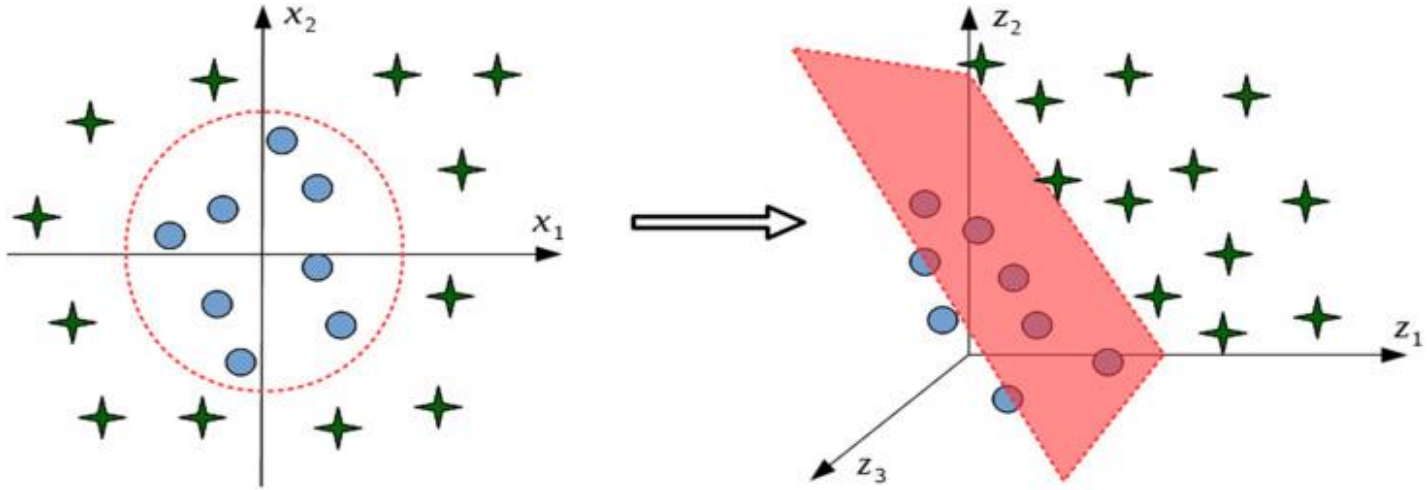
$$F1\ score = \frac{2}{\frac{1}{Precision} + \frac{1}{Recall}}$$

Other ML Models

- **Support Vector Machine (SVM):**

  Key Idea:
  - SVM finds the best boundary (hyperplane) that separates classes by maximizing the margin between them.
  - Works well with high-dimensional data and small datasets.
  - Can be computationally expensive for large datasets.



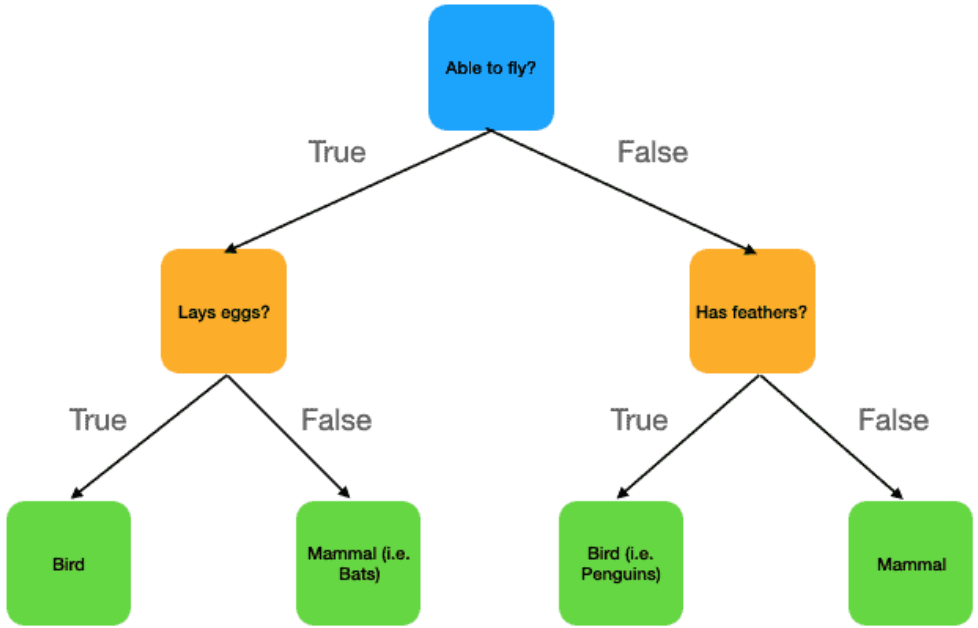


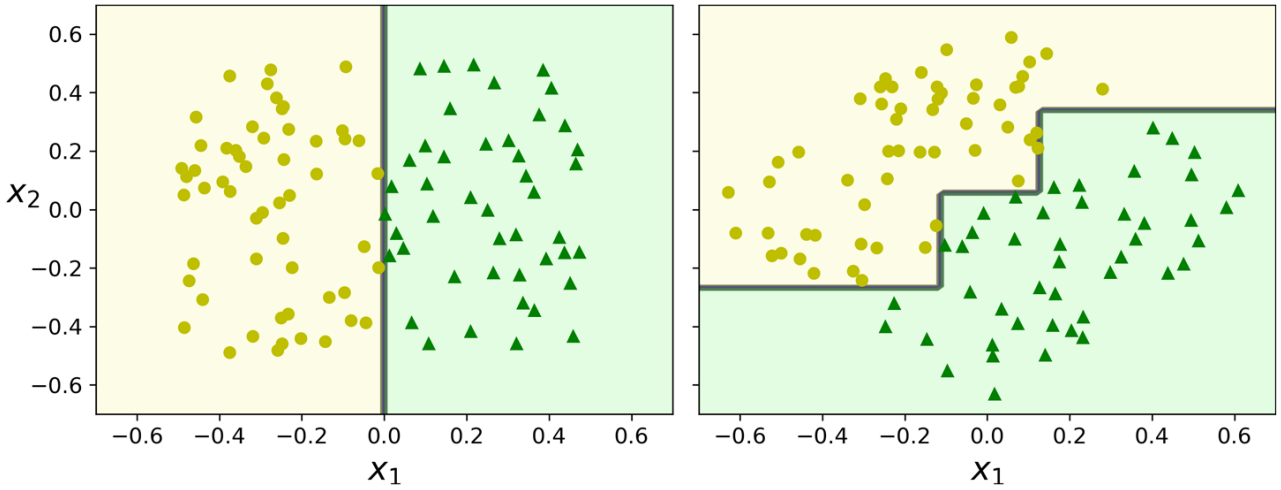
Other ML Models

- **Decision Trees:**

  Key Idea:

  - A tree-based model splits data into subsets based on feature thresholds, creating a flowchart-like structure for decision-making.
  - Overfits very easily.
  - could work bad if relation between features and the target is linear. (Check the right figure)





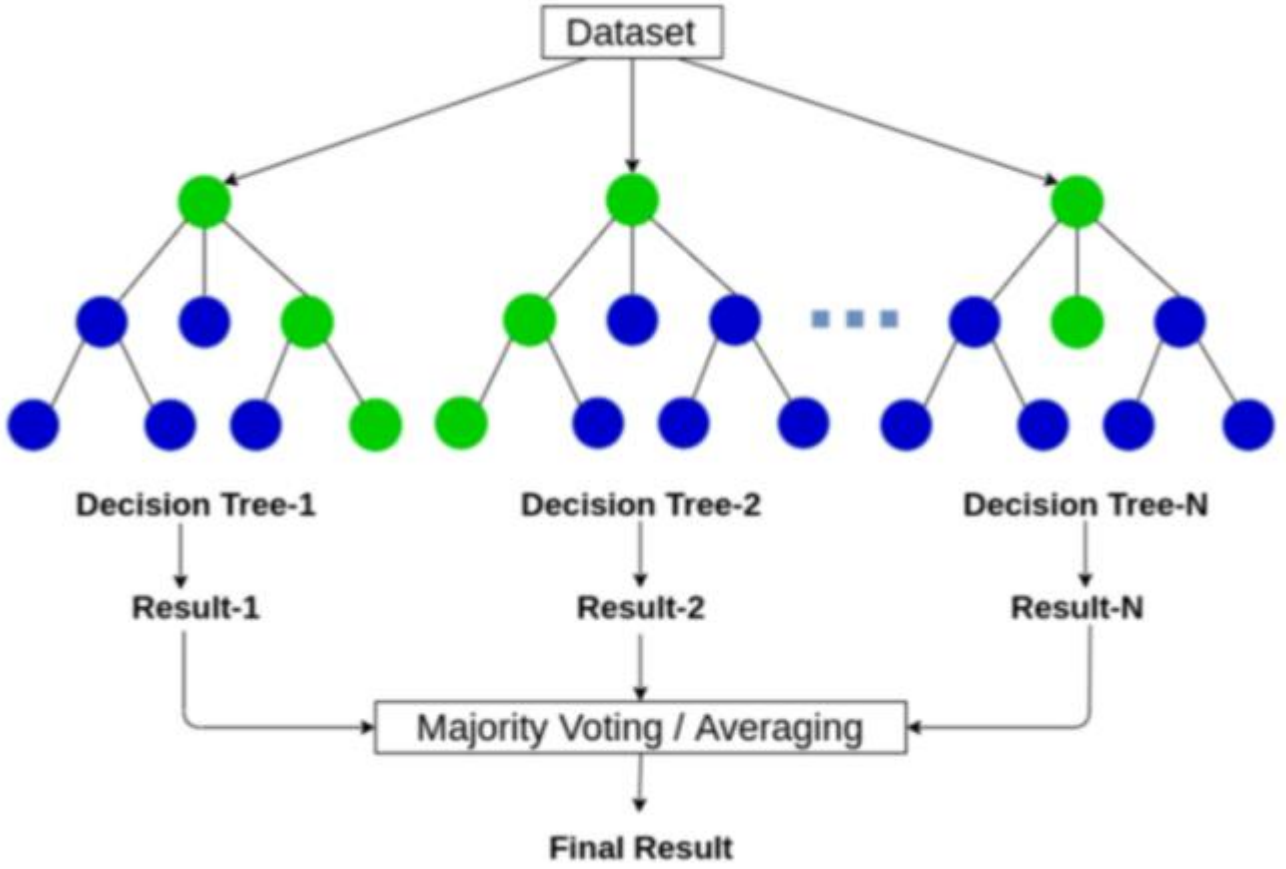
Other ML Models

- **Random Forest:**

  Instead of training one tree, let's train a forest :)

  Key Idea:

  - Training K decision trees <u>independently</u>, where each one is fed with a different subset of samples and features.
  - Then average the results (Regression) or Take the mode (Classification).
  - Powerful model.

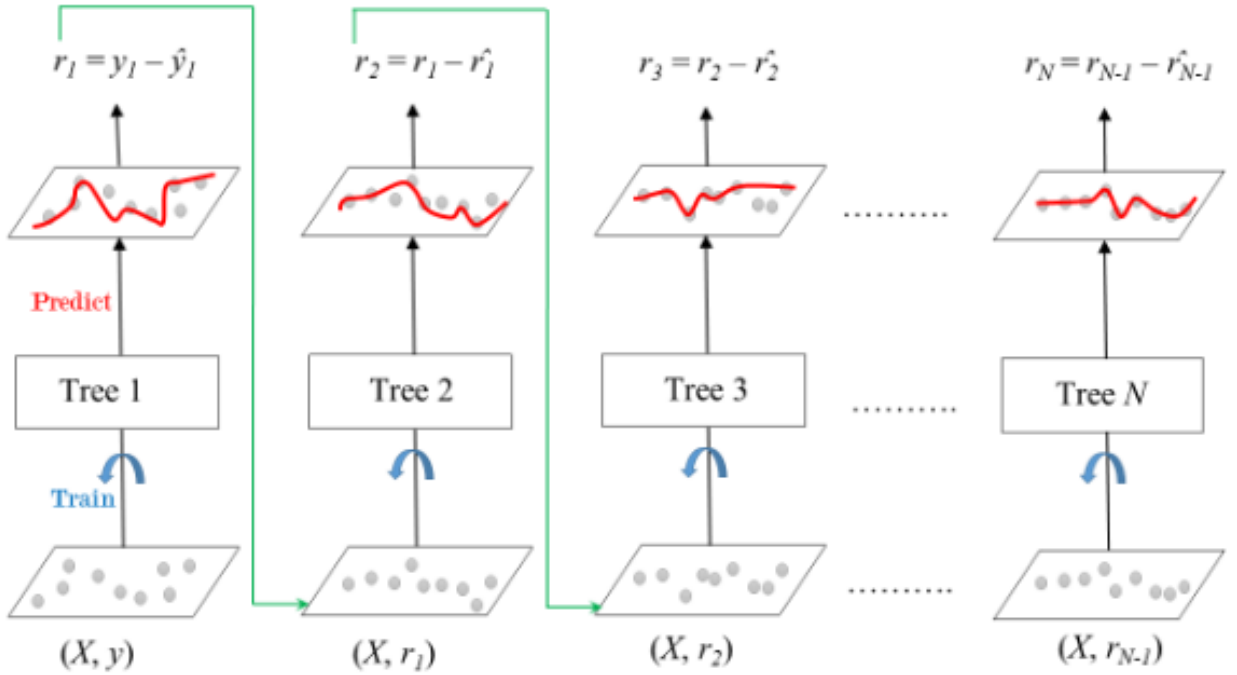


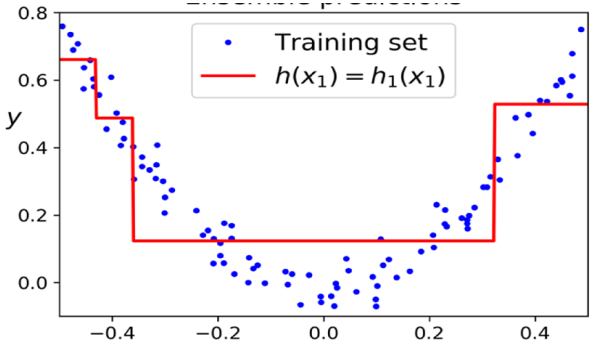
Other ML Models

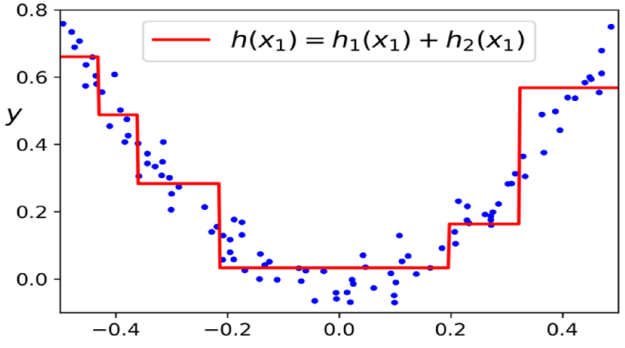
- **Gradient Boosting:**

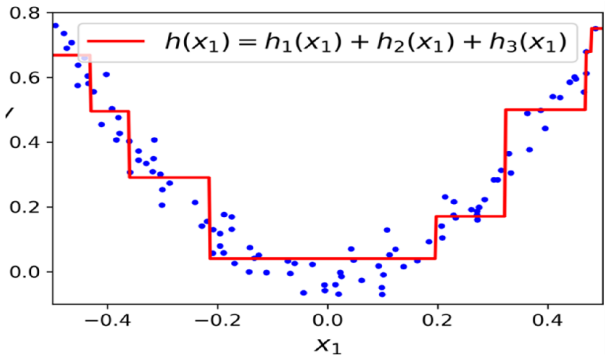
  Key Idea:

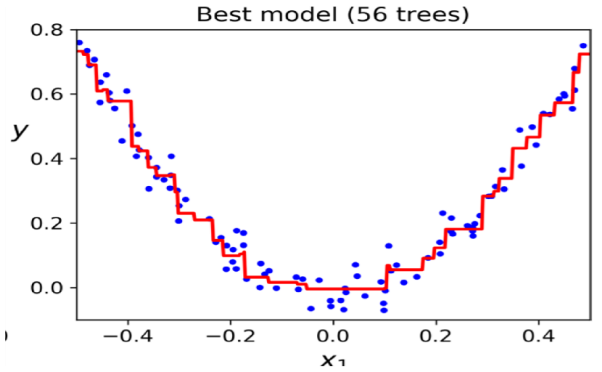
  - Training K decision trees <u>dependently</u>, where each tree corrects the errors of the previous one.
  - Many versions: XGBoost, LightGBM, CatBoost,…etc
  - Top ML models.











A Slight Detour: A Look at Optimization Tools

Introduction

- **Optimizers** are algorithms that adjust the weights of the model to minimize the loss function during training.
- They are one of the main components of Deep Learning models.
- We will go through several types of optimizers in this section.

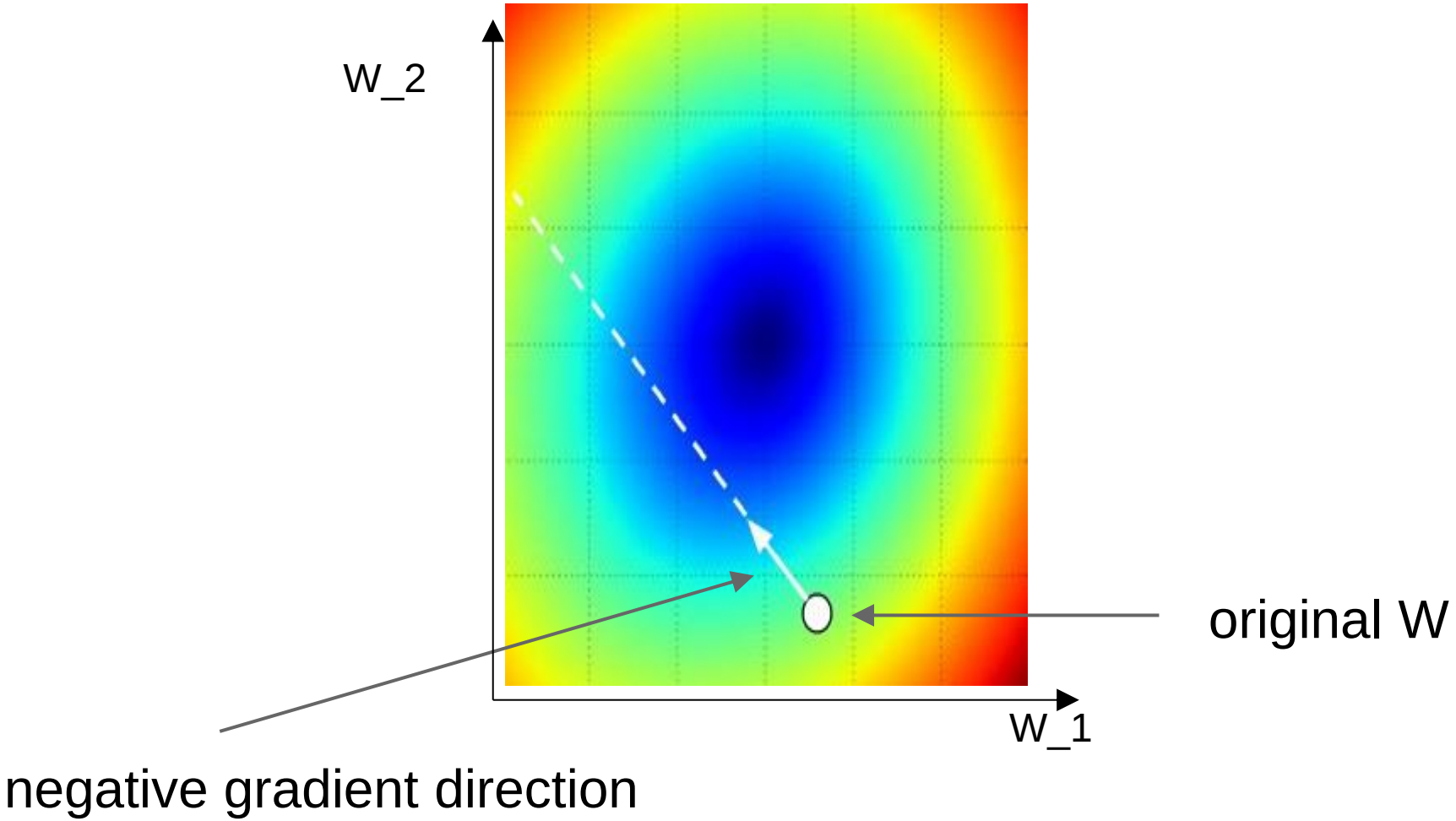
Direction of maximum increase and decrease for a function

- Gradient direction is the direction of maximum increase for a function
- Negative gradient is the direction of maximum decrease for a function

Gradient Descent

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```


W_2
W_1
original W
negative gradient direction

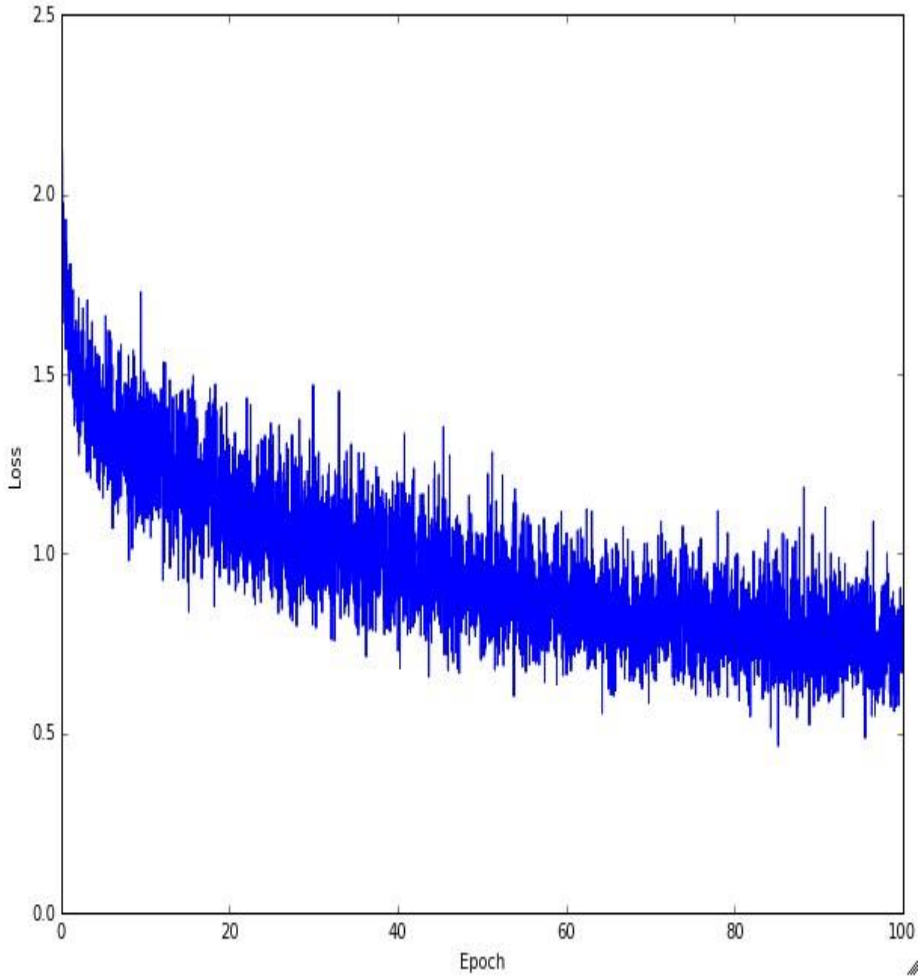
Mini-batch Gradient Descent

- only use a small portion of the training set to compute the gradient.

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

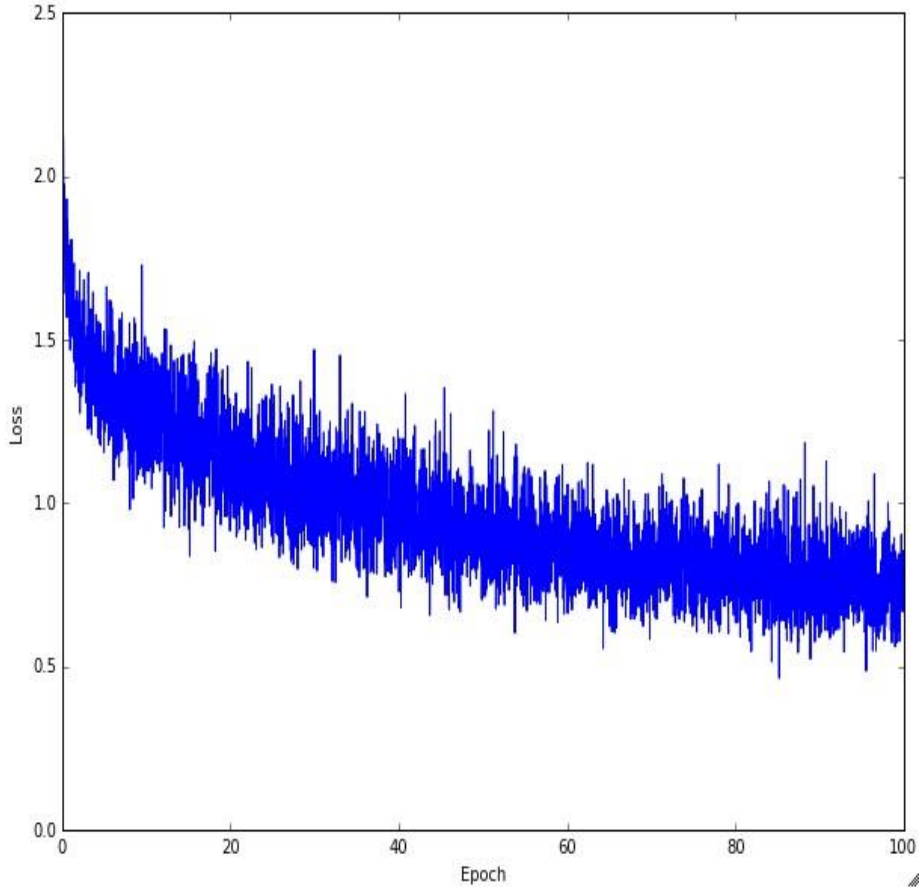
Common mini-batch sizes are 32/64/128 examples
e.g. Krizhevsky ILSVRC ConvNet used 256 examples

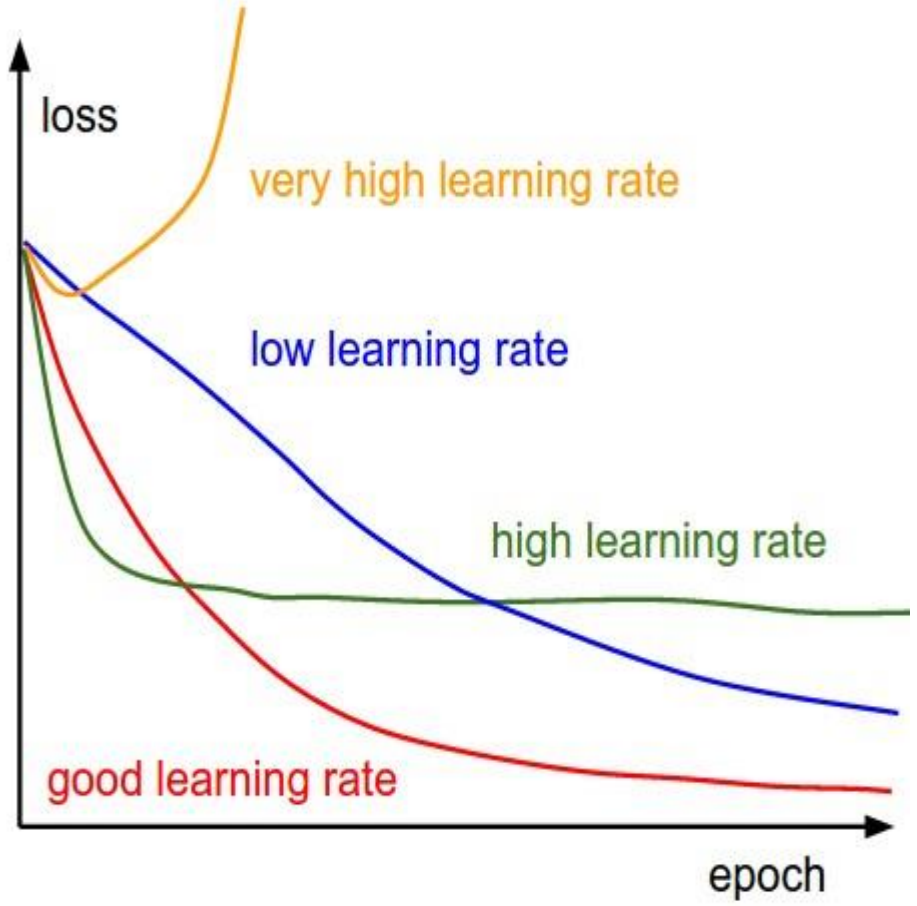


Example of optimization progress while training a neural network.

(Loss over mini-batches goes down over time.)

The effects of step size (or “learning rate”)





Slides based on cs231n by Fei-Fei Li & Andrej Karpathy & Justin Johnson

Mini-batch Gradient Descent

- only use a small portion of the training set to compute the gradient.

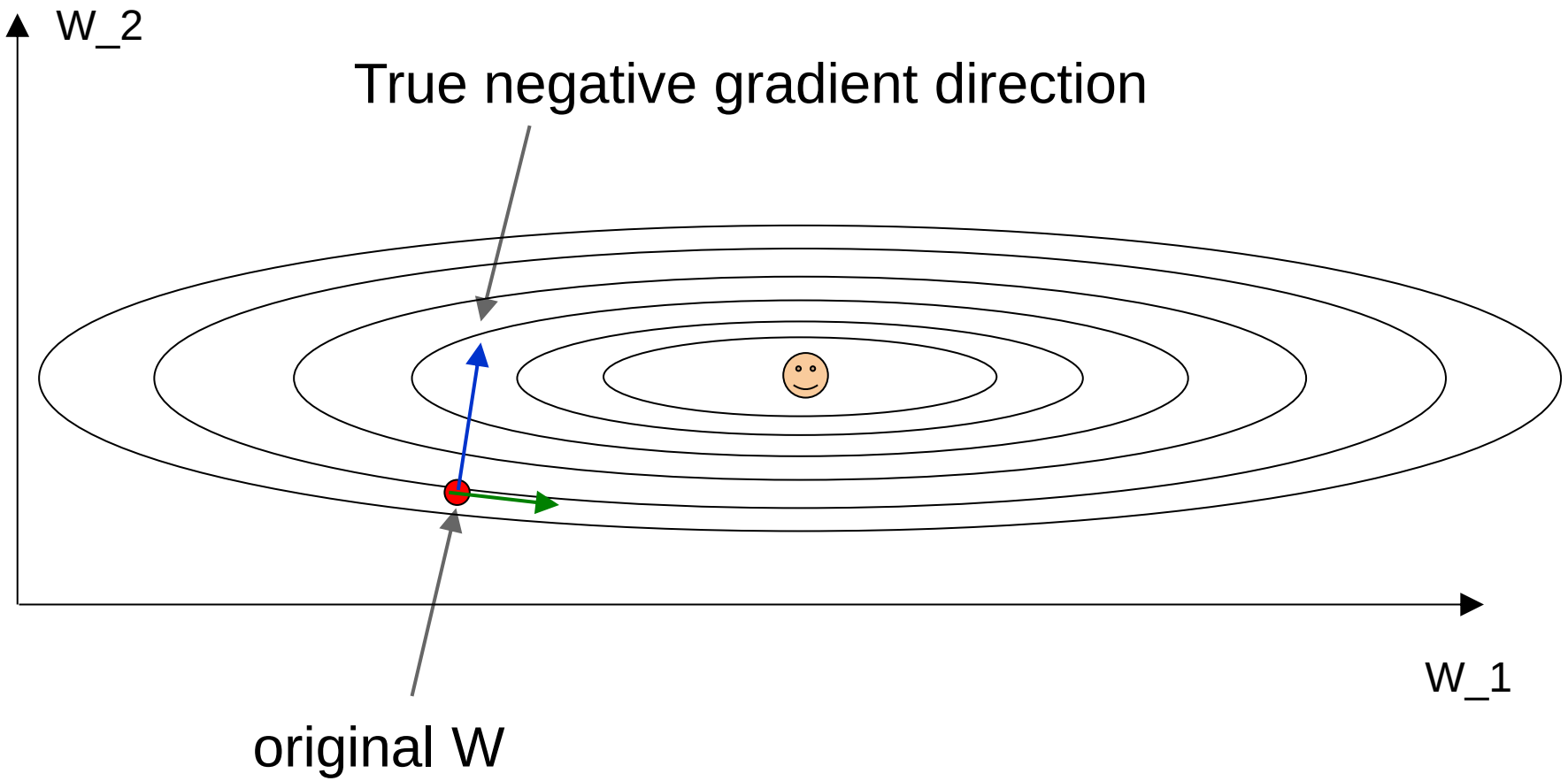
```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

we will look at more fancy update formulas (momentum, Adagrad, RMSProp, Adam, …)

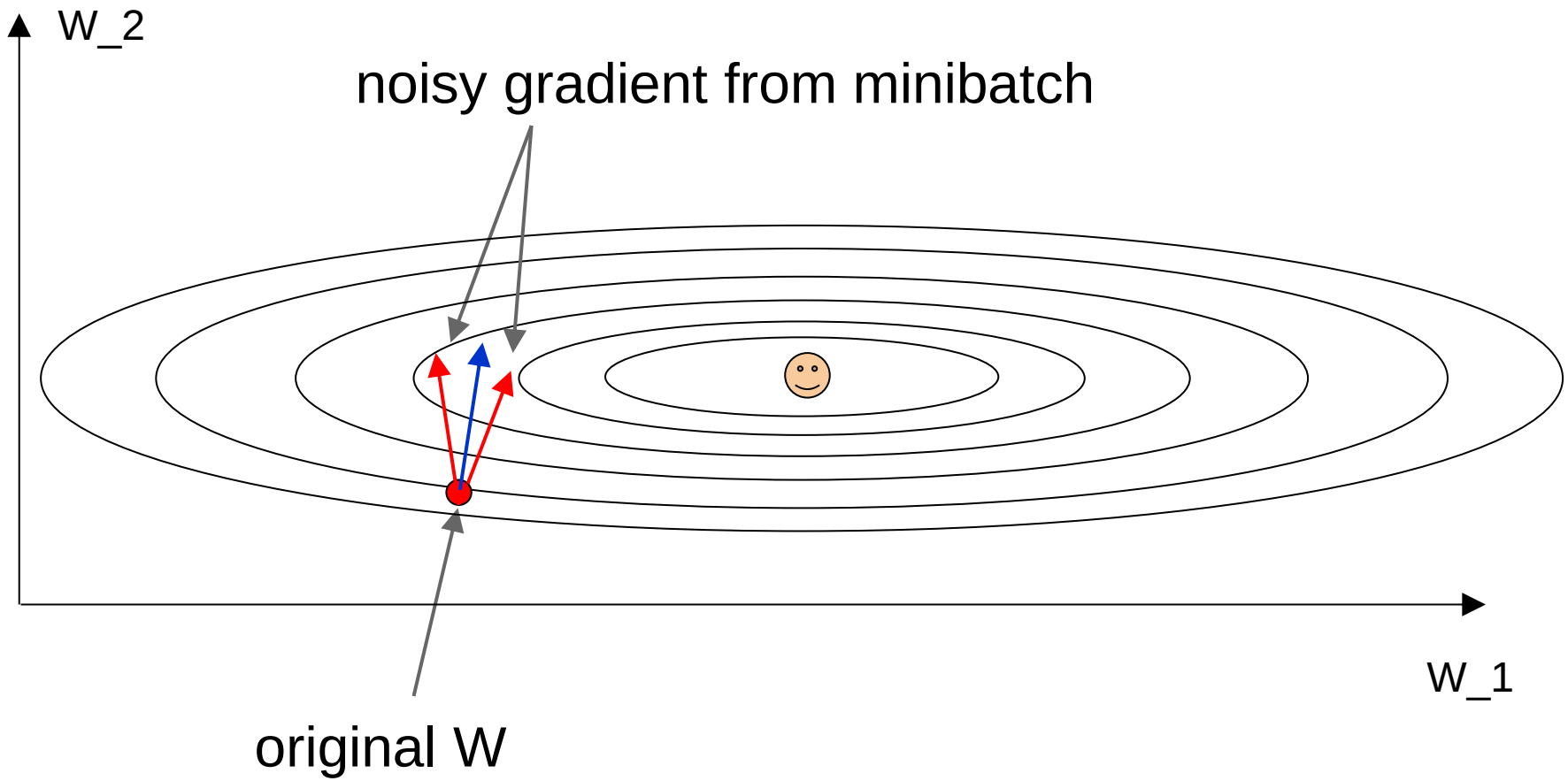
Common mini-batch sizes are 32/64/128 examples
e.g. Krizhevsky ILSVRC ConvNet used 256 examples

Minibatch updates



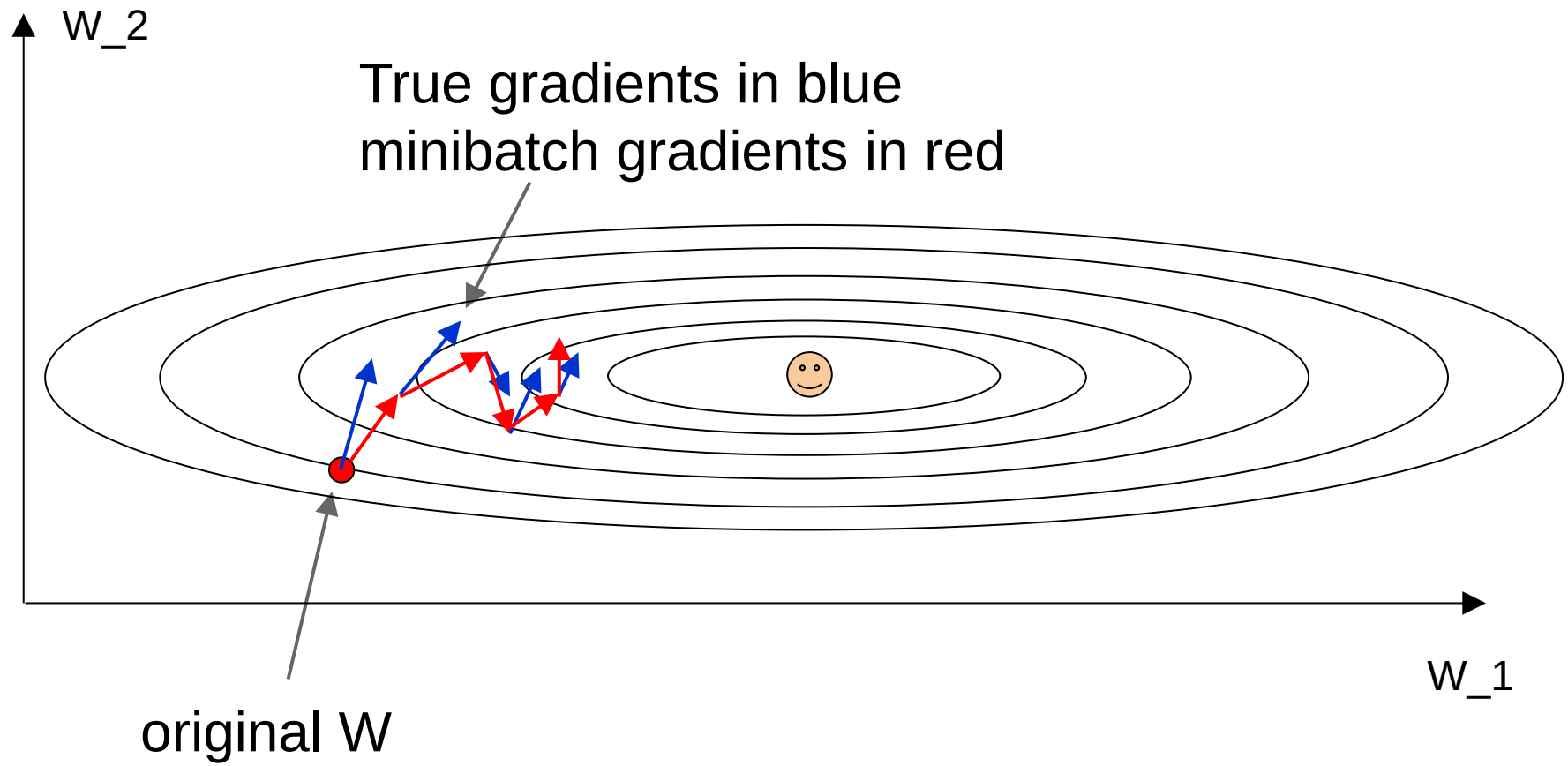
Slide based on CS294-129 by John Canny

Minibatch Gradient Descent



Slide based on CS294-129 by John Canny

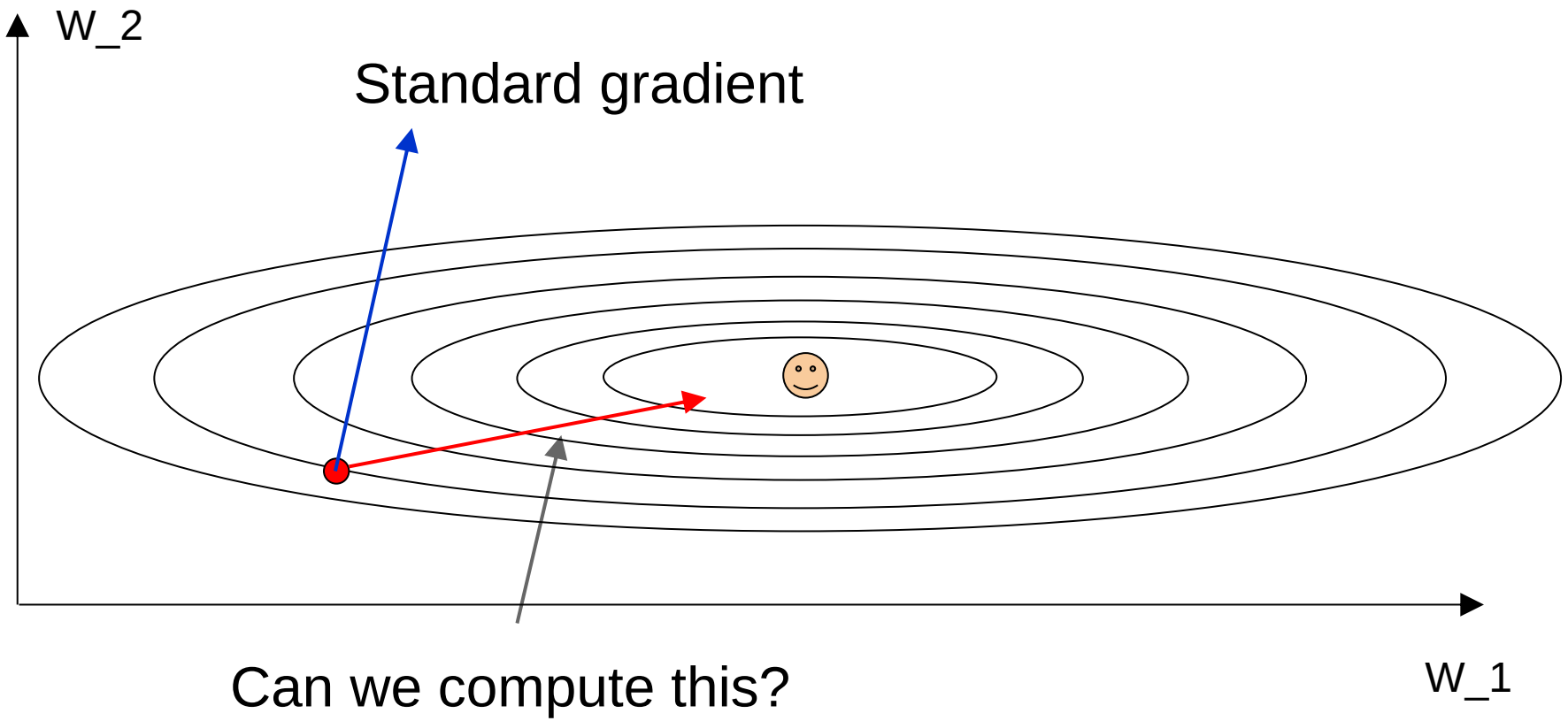
Minibatch Gradient Descent



Gradients are noisy but still make good progress on average

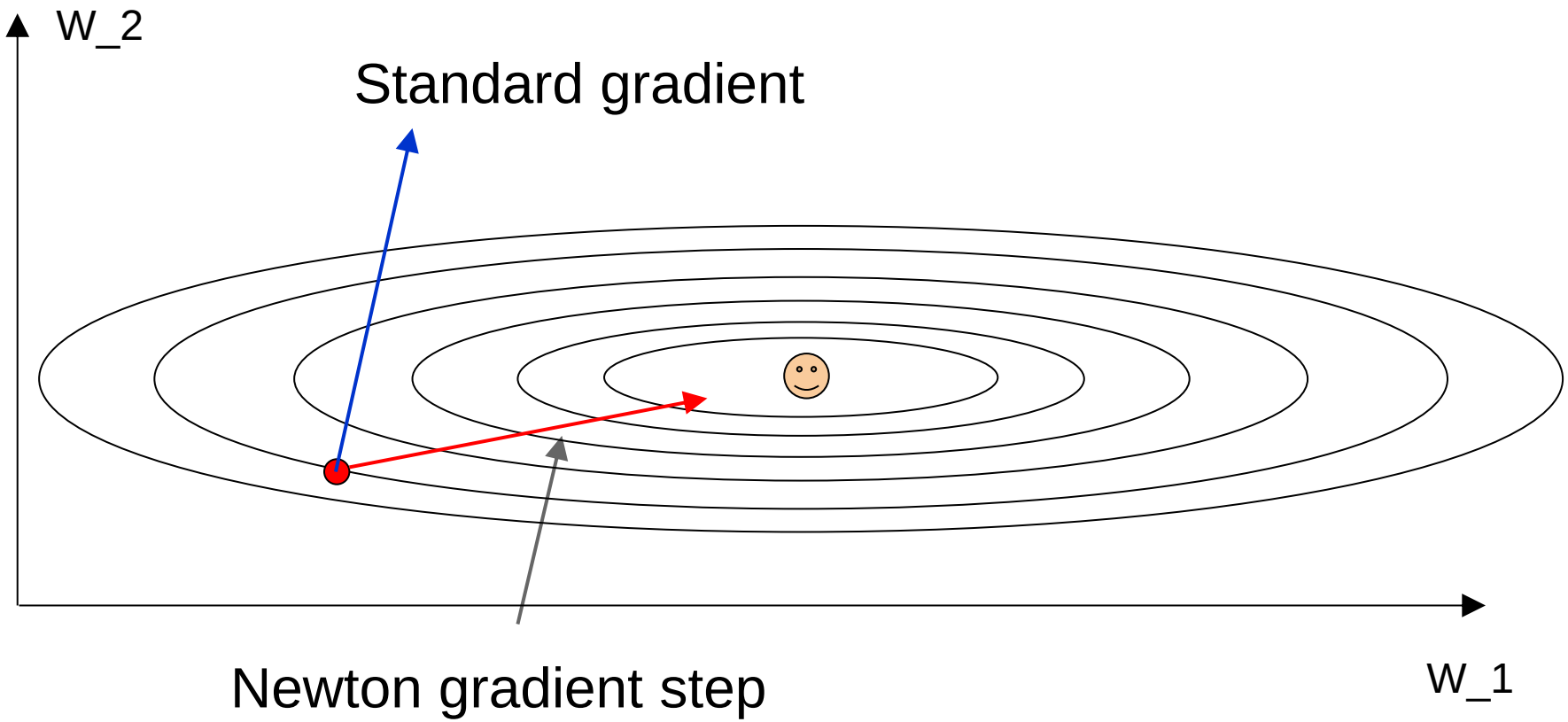
Slide based on CS294-129 by John Canny

You might be wondering…



Slide based on CS294-129 by John Canny

Newton step



Slide based on CS294-129 by John Canny

Newton's method for gradients:

The Newton update is:

$$x_{n+1} = x_n - H_f(x_n)^{-1}\nabla f(x_n)$$

Where H_f is the Hessian matrix of second derivatives of f.

Converges very fast, but rarely used in DL. Why?

Slide based on CS294-129 by John Canny

Newton's method for gradients:

The Newton update is:

$$x_{n+1} = x_n - H_f(x_n)^{-1} \nabla f(x_n)$$

Converges very fast, but rarely used in DL. Why?

Too expensive: if x_n has dimension M, the Hessian $H_f(x_n)$ has dimension M^2 and takes $O(M^3)$ time to invert.

Slide based on CS294-129 by John Canny

Newton's method for gradients:

The Newton update is:

$$x_{n+1} = x_n - H_f(x_n)^{-1} \nabla f(x_n)$$

Converges very fast, but rarely used in DL. Why?

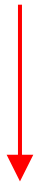
Too expensive: if x_n has dimension M, the Hessian $H_f(x_n)$ has dimension M^2 and takes $O(M^3)$ time to invert.

Too unstable: it involves a high-dimensional matrix inverse, which has poor numerical stability. The Hessian may even be singular.

Slide based on CS294-129 by John Canny

Momentum update

```
# Gradient descent update
x += - learning_rate * dx
```

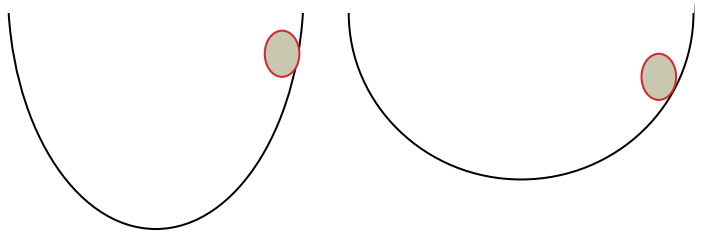


```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

- Physical interpretation as ball rolling down the loss function + friction (mu coefficient).
- mu = usually ~0.5, 0.9, or 0.99 (Sometimes annealed over time, e.g. from 0.5 -> 0.99)

Momentum update

```
# Gradient descent update
x += - learning_rate * dx
```



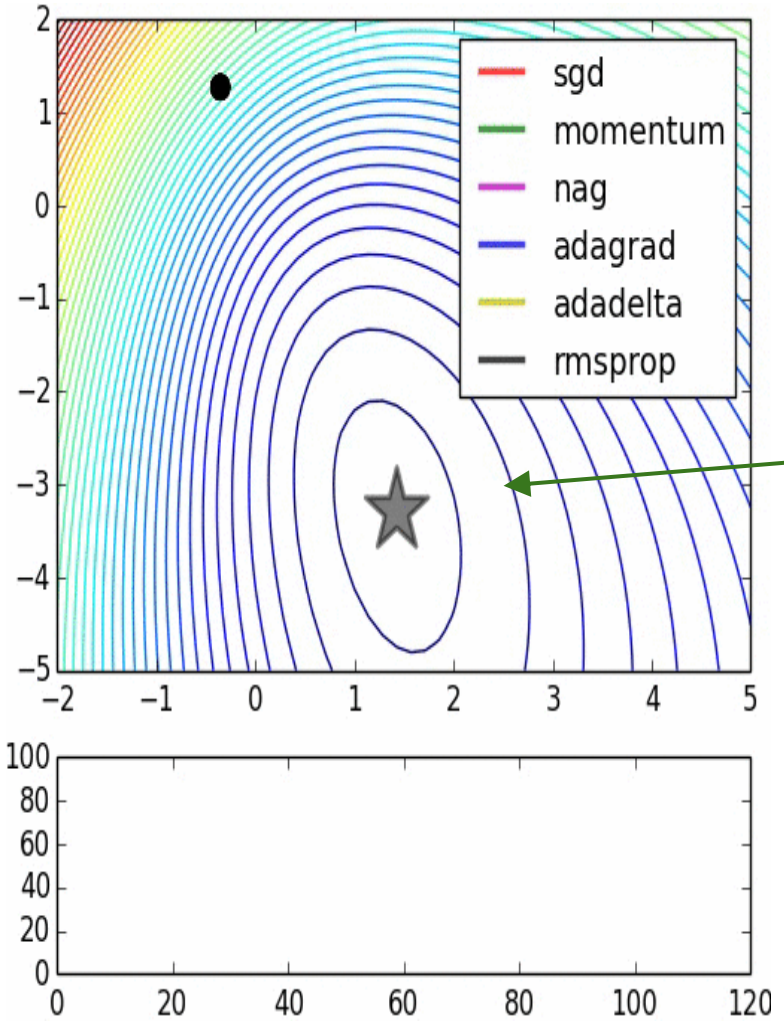
```
# Momentum update
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

- Allows a velocity to “build up” along shallow directions
- Velocity becomes damped in steep direction due to quickly changing sign

SGD

vs

Momentum



notice momentum overshooting the target, but overall getting to the minimum much faster than vanilla SGD.

AdaGrad update

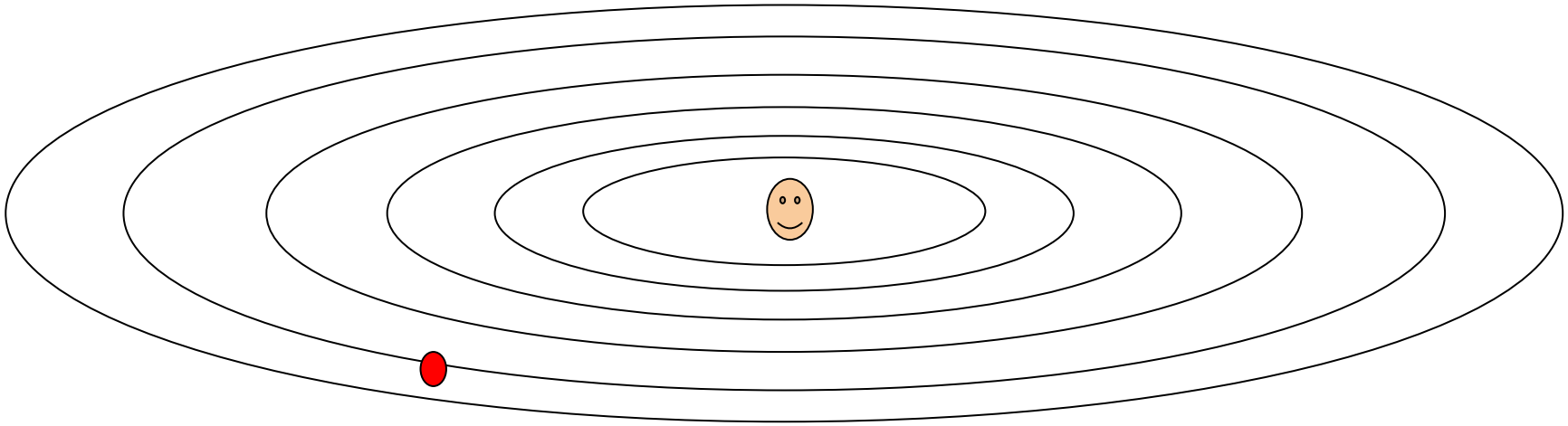
[Duchi et al., 2011]


```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

AdaGrad update

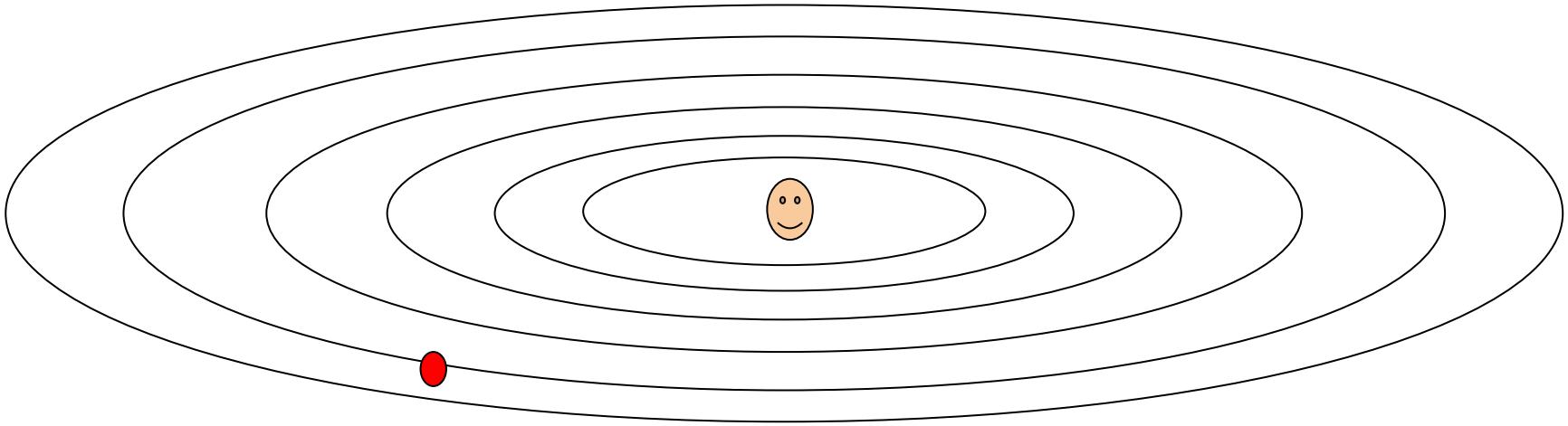
```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



Q: What happens with AdaGrad?

AdaGrad update

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



Q2: What happens to the step size over long time?

RMSProp update

[Tieleman and Hinton, 2012]

```
# Adagrad update
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

```
# RMSProp
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

rmsprop: A mini-batch version of rprop

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
  - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight

$$MeanSquare(w,t) = 0.9\, MeanSquare(w,\, t-1) + 0.1 \left(\partial E / \partial w (t) \right)^2$$

- Dividing the gradient by $\sqrt{MeanSquare(w,\, t)}$ makes the learning work much better (Tijmen Tieleman, unpublished).

Introduced in a slide in Geoff Hinton's Coursera class, lecture 6

rmsprop: A mini-batch version of rprop

- rprop is equivalent to using the gradient but also dividing by the size of the gradient.
  - The problem with mini-batch rprop is that we divide by a different number for each mini-batch. So why not force the number we divide by to be very similar for adjacent mini-batches?
- rmsprop: Keep a moving average of the squared gradient for each weight

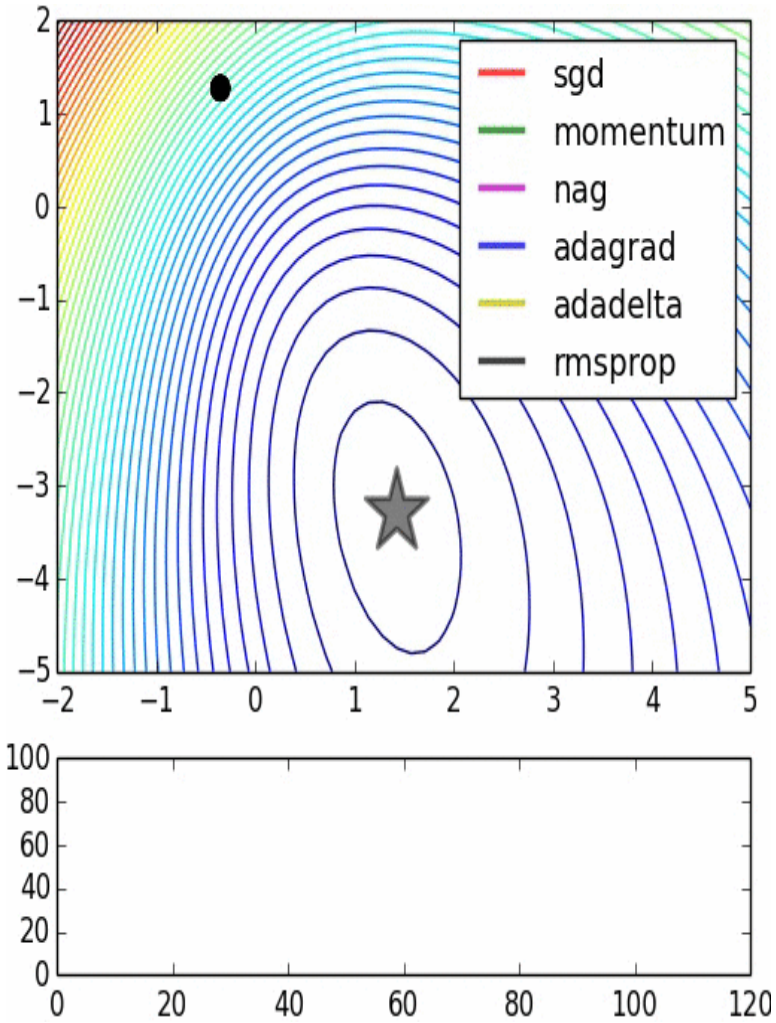
$$MeanSquare(w,t) = 0.9\, MeanSquare(w,\ t-1) + 0.1\left(\partial E/\partial w(t)\right)^2$$

- Dividing the gradient by $\sqrt{MeanSquare(w,\ t)}$ makes the learning work much better (Tijmen Tieleman, unpublished).

Introduced in a slide in Geoff Hinton's Coursera class, lecture 6

Cited by several papers as:

[52] T. Tieleman and G. E. Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude., 2012.



adagrad
rmsprop

Adam update

[Kingma and Ba, 2014]

(incomplete, but close)

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

Adam update

[Kingma and Ba, 2014]

(incomplete, but close)

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

momentum

RMSProp-like

Looks a bit like RMSProp with momentum

Adam update

[Kingma and Ba, 2014]

(incomplete, but close)

```
# Adam
m = beta1*m + (1-beta1)*dx # update first moment
v = beta2*v + (1-beta2)*(dx**2) # update second moment
x += - learning_rate * m / (np.sqrt(v) + 1e-7)
```

momentum

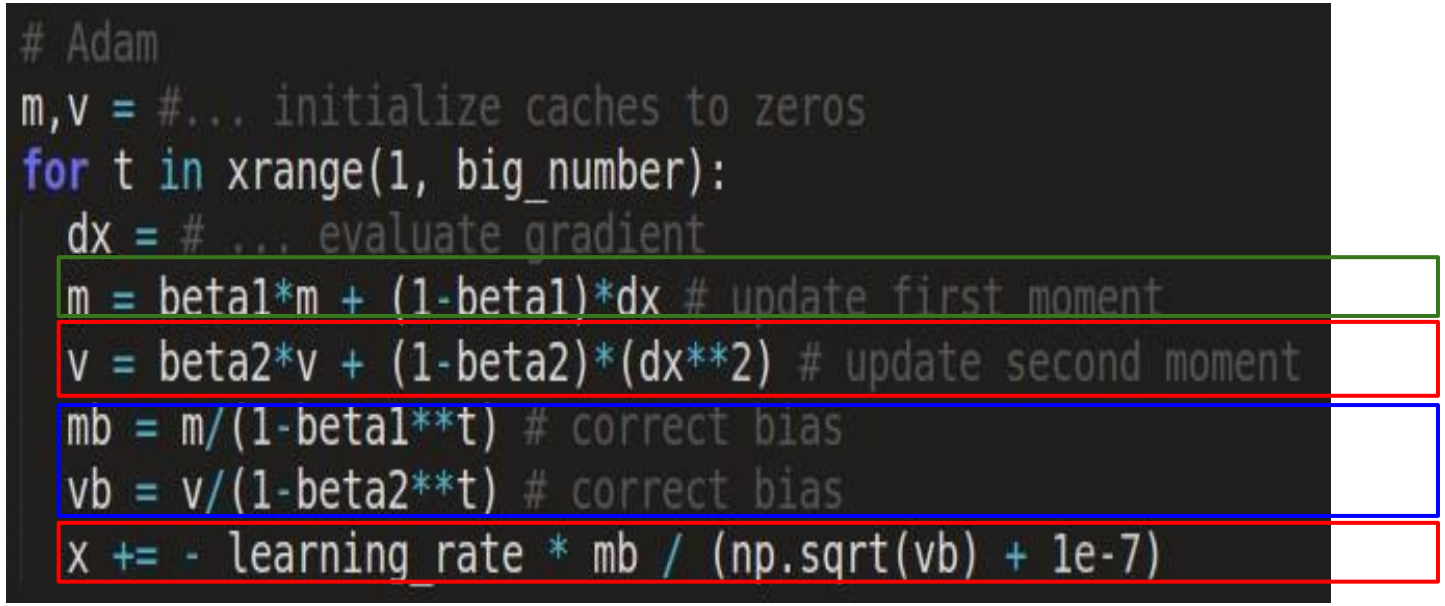
RMSProp-like

Looks a bit like RMSProp with momentum

```
# RMSProp
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

Adam update

[Kingma and Ba, 2014]



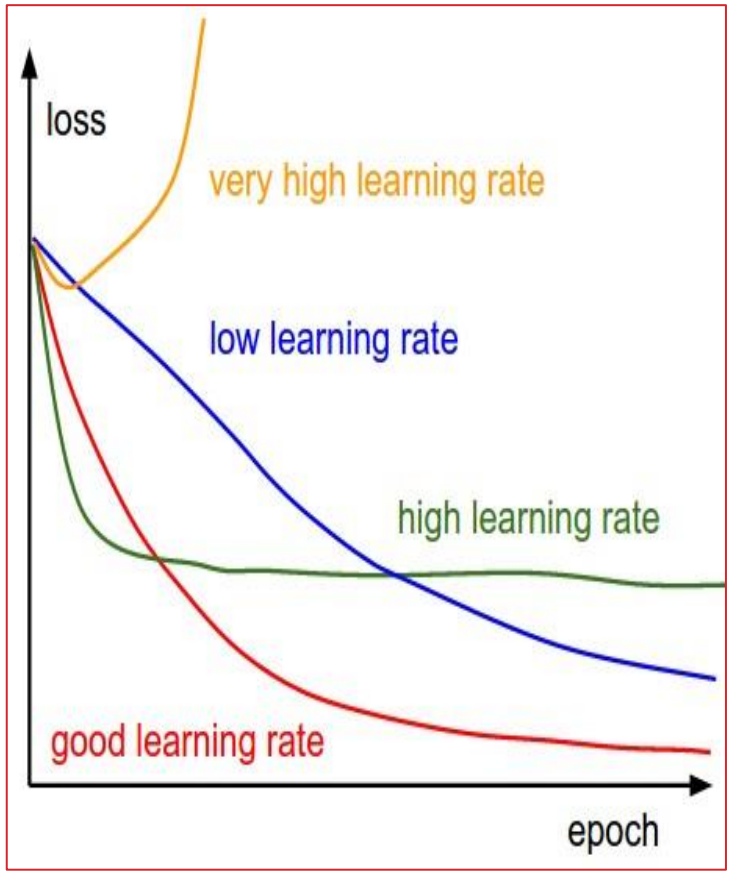
momentum

bias correction
(only relevant in first few iterations when t is small)

RMSProp-like

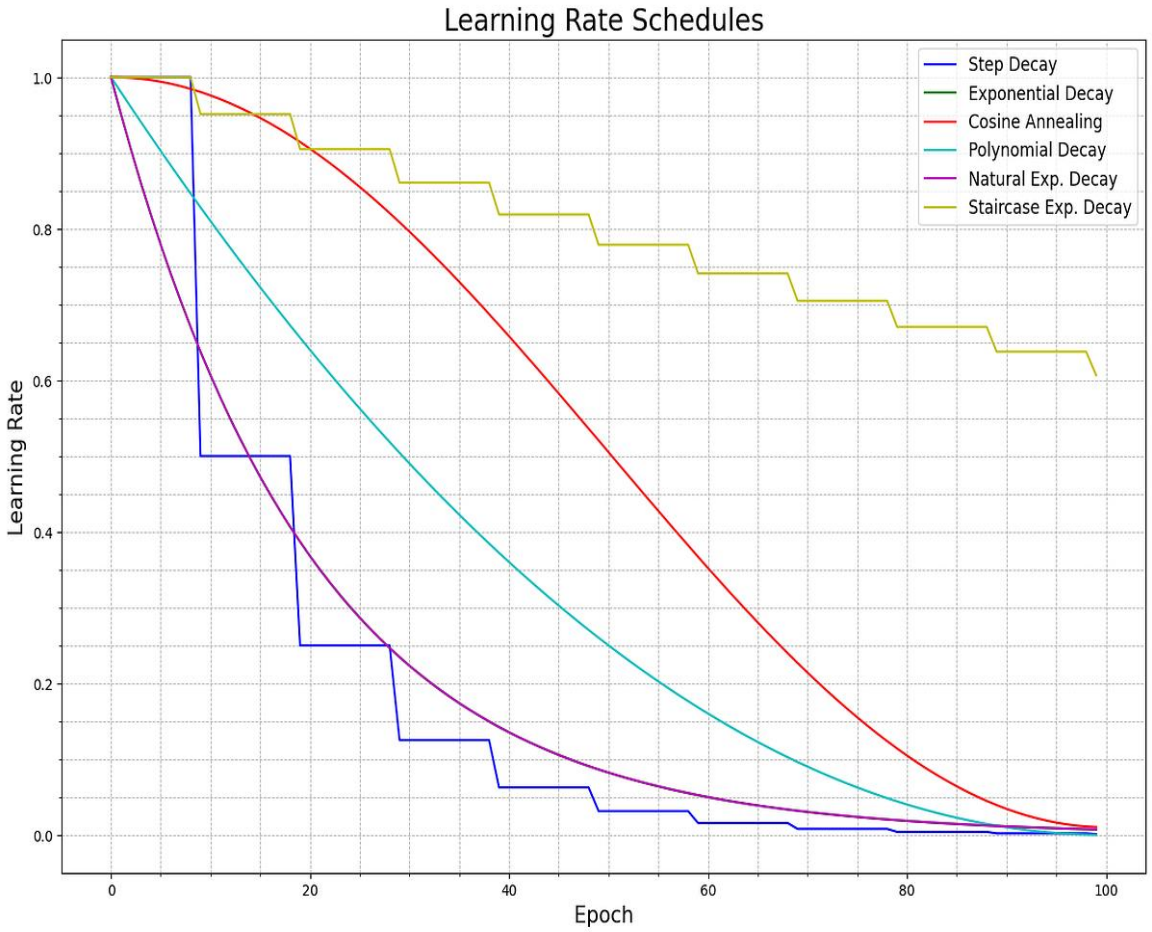
The bias correction compensates for the fact that m,v are initialized at zero and need some time to “warm up”.

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



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

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



⇒ **Learning rate decay over time (LR Schedulers)!**

Step decay:
e.g. decay learning rate by half every few epochs.

Exponential decay:

$$\alpha = \alpha_0 e^{-kt}$$

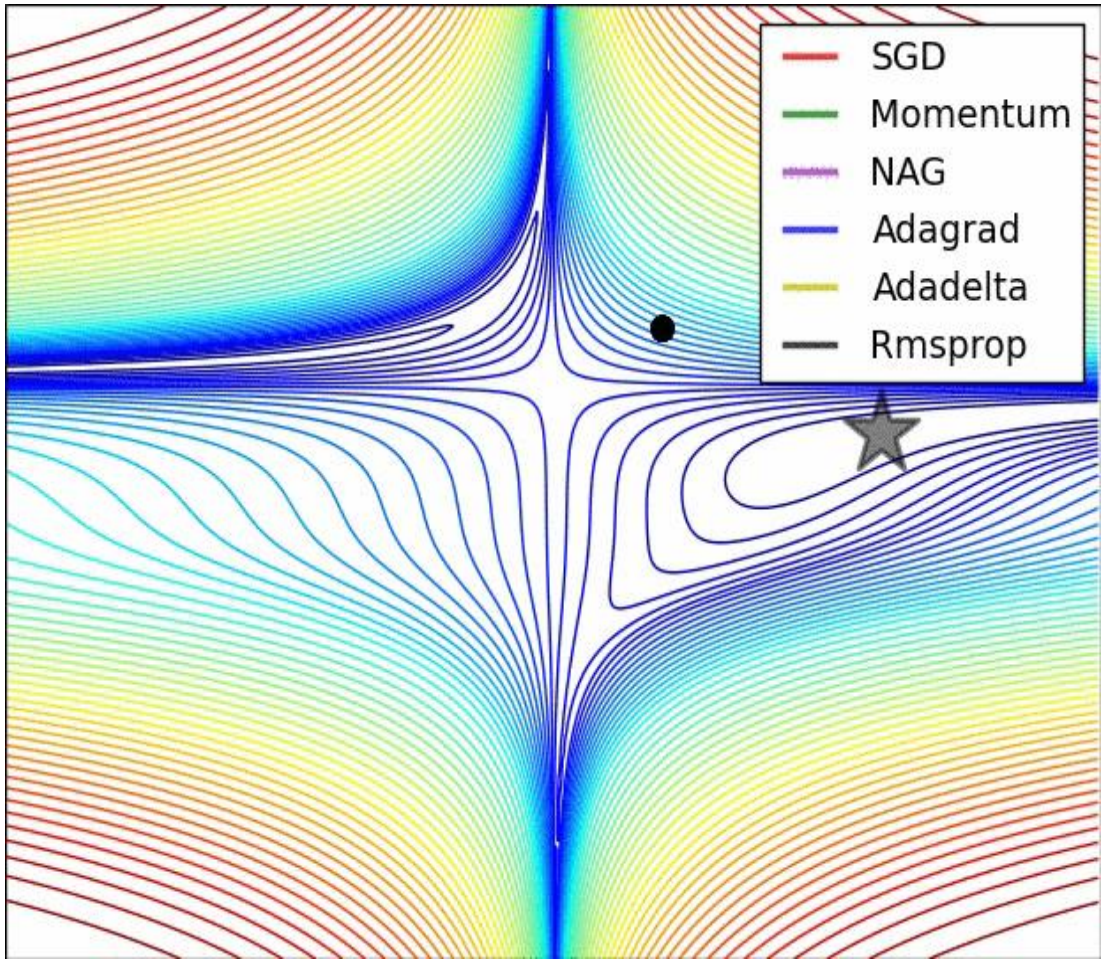
Cosine decay (Best):

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})\left(1 + \cos\left(\frac{T_{cur}}{T_{max}}\pi\right)\right)$$

Slides based on cs231n by Fei-Fei Li & Andrej Karpathy & Justin Johnson

Summary

- **Simple Gradient Methods** like SGD can make adequate progress to an optimum when used on minibatches of data.
- **Second-order** methods make much better progress toward the goal, but are more expensive and unstable.
- **Momentum:** is another method to produce better effective gradients.
- **ADAGRAD, RMSprop** diagonally scale the gradient.
- **ADAM** scales and applies momentum.



(image credits to Alec Radford)

Questions?