Introduction

- Optimisation technique
- Popular for tuning neural networks
- Previously used in MaxEnt, CRF, …
- Goal:
  - Minimise an objective function $J(\theta)$
  - $\theta \in R^d$ are a model's parameters
  - Update params in opposite direction to gradient $\nabla_\theta J$
  - Learning rate η determines step size

Terminology

- x : examples
- y : labels
- J : loss function (how bad the mistakes are)

  e.g. training error; the error on the training data
- θ : parameters
- η : rate

Three variants

- Batch gradient descent
  - Vanilla, original
  - Compute loss over whole training dataset at a time
- Stochastic gradient descent
  - Loss measured per example & label
- Mini-batch gradient descent
  - “Best of both worlds”
  - Loss computed over subset of training dataset

Batch gradient descent

- Computes gradient of cost function w.r.t. to parameters θ for the entire training dataset:
  - $\theta=\theta-\eta \ \nabla_\theta J(\theta)$
- Need to calculate the gradients for the whole dataset to perform just one update
  - Batch gradient descent can be slow and is intractable for datasets that don't fit in memory
  - Batch gradient descent also doesn't allow us to update our model online, i.e. with new examples on-the-fly.

Batch gradient descent

- Code might look like
  - for i in range(nb_epochs):
  -   params_grad = evaluate_gradient(loss_function, data, params)
  -   params = params - learning_rate * params_grad
- Given a set number of training epochs:
  - Compute the gradient vector of the loss function for the whole dataset, w.r.t. our parameter vector params
  - Update our parameters in the opposite direction of the gradients
  - Learning rate η determines update size

Batch gradient descent

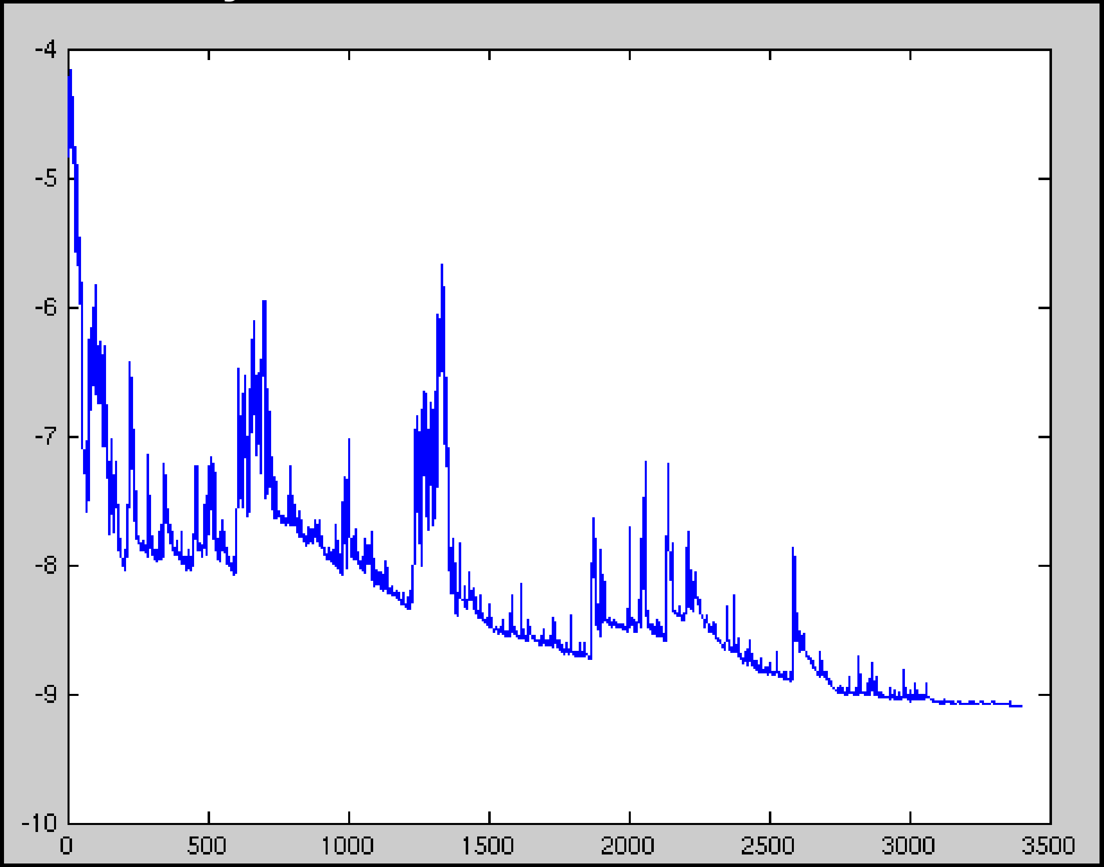
- Guaranteed to:
  - Converge to the global minimum for convex error surfaces
  - Converge to a local minimum for non-convex surfaces.

Stochastic gradient descent

- Stochastic = Random
- SGD performs a parameter update for each training example $x^{(i)}$ and label $y^{(i)}$:
  - $\theta=\theta-\eta \quad \nabla_\theta J(\theta;x^{(i)};y^{(i)})$
- Batch GD does redundant work:
  - Recomputes gradients for similar examples before doing parameter update
- SGD more efficient:
  - One update at a time
  - What advantage does this give?
  - Possible to apply to online learning

Stochastic gradient descent

- SGD performs frequent updates:
  - high variance
  - Causes objective function to fluctuate heavily



Stochastic gradient descent

- Other advantages of SGD vs. batch?
  - enables it to jump to new and potentially better local minima
- Side-effect of SGD’s approach to finding minima?
  - Complicates convergence: SGD will keep overshooting
- Solution for this?
  - Slowly decrease learning rate

Stochastic gradient descent

- Example code:

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

- Similar behaviour to batch GD:
  - Almost certainly converging to a local minimum for non-convex optimisation
  - Almost certainly converging to a global minimum for convex optimisation

Mini-batch gradient descent

- Aims to take best of both worlds
- Performs update for every mini-batch of *n* examples:
  - $\theta = \theta - \eta \quad \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$
- Reduces variance of parameter updates
  - Leads to more stable convergence
- Can use some neat optimisations
  - E.g. GPU matrix factorisation
- Algorithm of choice for NN training

Mini-batch gradient descent

- For n=50:

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

- This updates parameters based on a J measured from a random subset of 50 examples

Mini-batch gradient descent

- Does not guarantee good convergence
- Choosing a proper learning rate can be difficult
  - Too small: very slow convergence
  - Too high:
    - Damages convergence
    - Loss rate can fluctuate or even diverge
- Learning rate “schedules” reduce η at preset times, or when objective delta falls below threshold

Mini-batch gradient descent

- Not all features have the same variance
  - A universal learning rate doesn’t make sense here
  - Do bigger updates for sparser features?

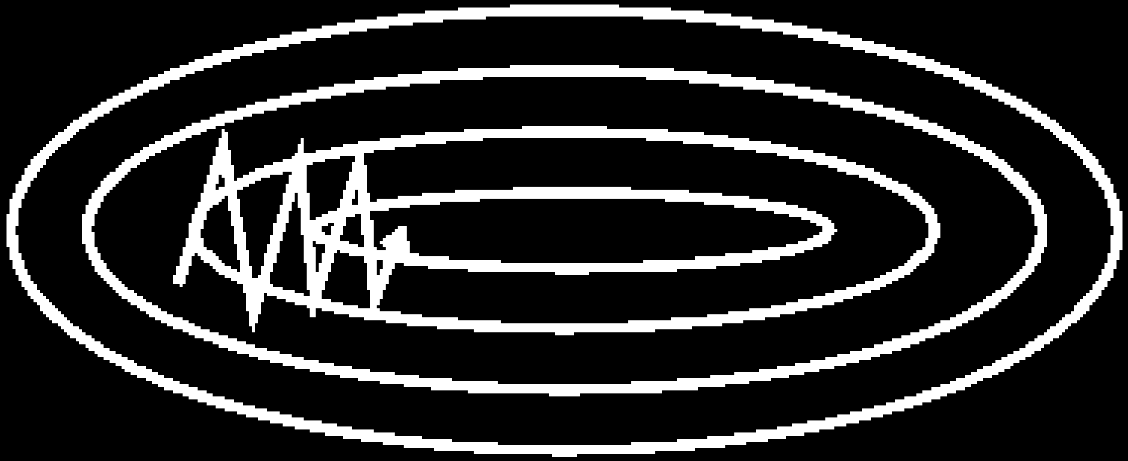
- Saddle points are a challenge
  - Neural nets tend to have many local minima
  - These tend to meet in saddle points
  - Equates to a plateau of the same error
  - Hard to escape using gradient-based method, as gradient $\rightarrow$ 0 in all dimensions

Gradient descent optimization

- Momentum
- Adagrad
- Adadelta
- RMSprop

Momentum optimisation

- SGD has trouble navigating "ravines"
  - areas where the surface curves much more steeply in one dimension than in another
  - Ravines more common than local optima
- SGD will oscillate across slopes while making poor progress to the optimum



Momentum optimisation

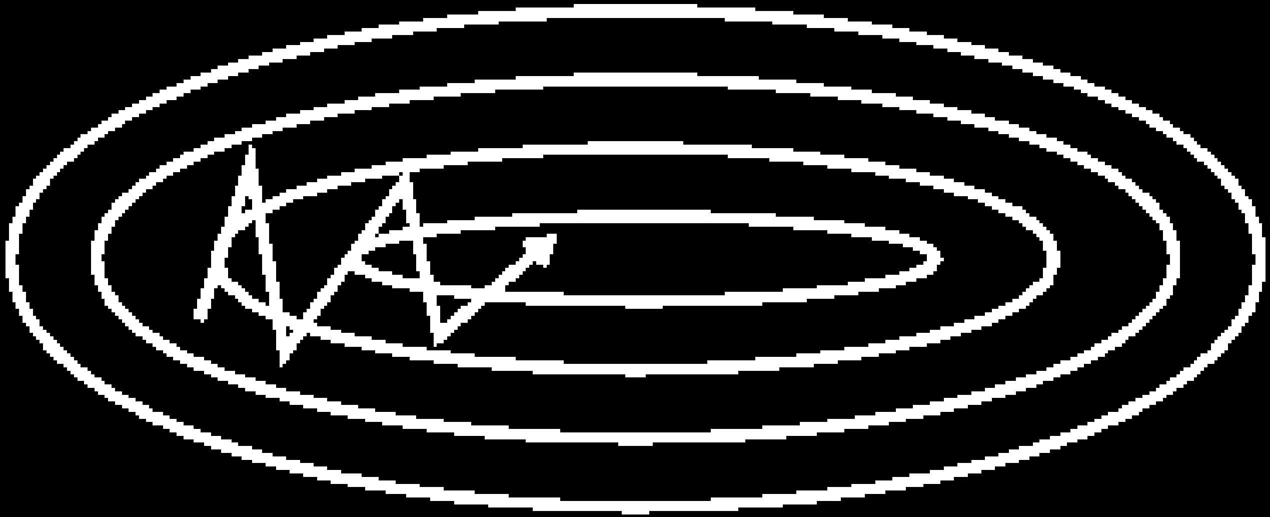
- Helps accelerate SGD in relevant direction
- Dampens oscillations
- Achieved by adding part γ of the last update to the current update
  - $v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$
  - $\theta = \theta - v_t$
- Parameter γ is the momentum

Momentum optimisation

- Using momentum is like pushing a ball down a hill.
  - The ball accelerates as it rolls downhill, becoming faster and faster..
  - ..until it reaches its terminal velocity if there is air resistance, i.e. $\gamma<1$
- The momentum term:
  - increases for dimensions whose gradients point in the same directions
  - reduces updates for dimensions whose gradients change directions.
- This gives faster convergence & reduced oscillation.

Momentum optimisation

- An example of momentum optimisation
  - Slows changes in high-gradient dimensions
  - Accelerates change in shallower dimensions



- Now we adapt to error function, why not adapt to parameter importance?

Adam optimisation

- Adaptive Moment Estimation (Adam) computes adaptive learning rates for each parameter.
- It stores two factors:
  - exponentially decaying average of past squared gradients v_t
  - exponentially decaying average of past gradients m_t, similar to momentum

$$m_t = \beta_1 m_t - 1 + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_t - 1 + (1 - \beta 2) g_t^2$$

Adam optimisation

- m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients
- m_t and v_t are initialized as vectors of 0's
  - they are biased towards zero,
  - especially during the initial time steps,
  - especially when the decay rates are small (i.e. β_1 and β_2 are close to 1).
- How can we avoid this?
  - Correct for bias in m and v

Adam optimisation

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}.$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}.$$

- Big beta will magnify *m* and *v*

Adam optimisation

- The adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t$$

- Typical values:
  - 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ

Visualisation of algorithms

- http://i.imgur.com/VkTJVVX.gif
- Adagrad, Adadelta, and RMSprop almost immediately head off in the right direction and converge similarly fast
- Momentum and NAG are led off-track, evoking the image of a ball rolling down the hill.

Visualisation at a saddle

- http://i.imgur.com/1Awcohc.gif
- SGD, Momentum, and NAG find it difficult to break symmetry, although the two latter eventually manage to escape the saddle point
- Adagrad, RMSprop, and Adadelta quickly head down the negative slope