NEURAL NETWORKS

LEARNING AND EVALUATION

BATCH NORMALISATION

BATCH NORMALISATION

- Data Preprocessing

BATCH NORMALISATION

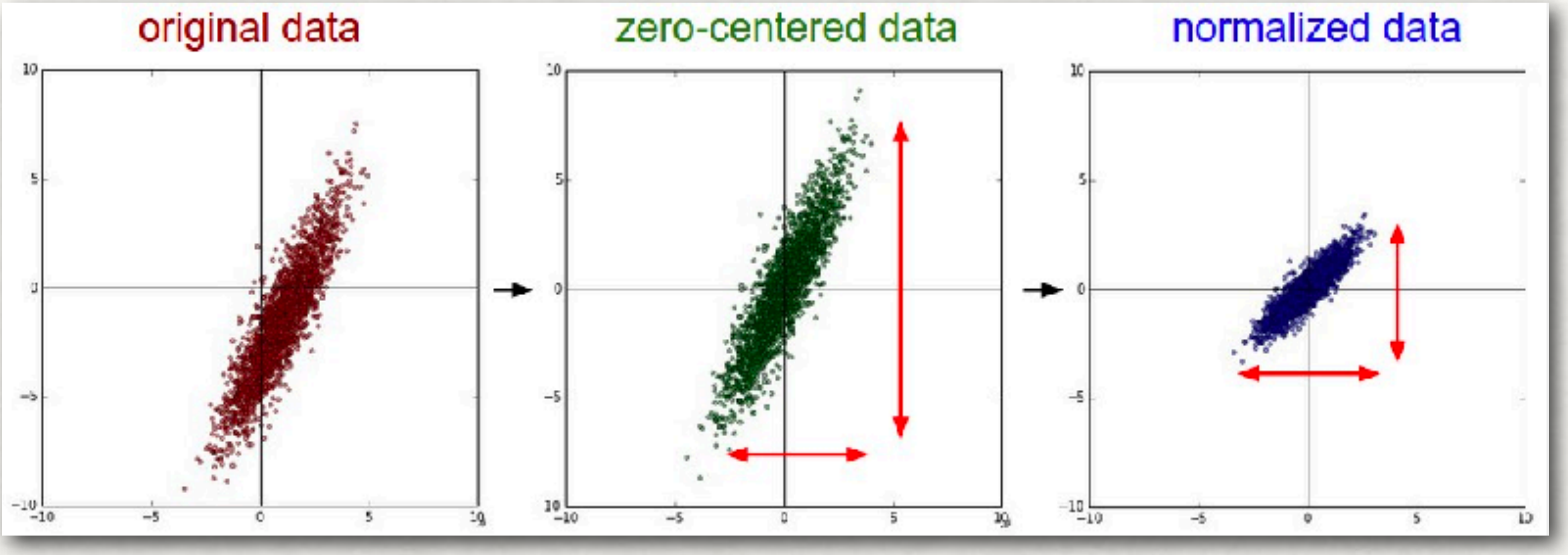
- Data Preprocessing
- Subtraction of the mean, followed by division with standard division

BATCH NORMALISATION

- Data Preprocessing
- Subtraction of the mean, followed by division with standard division
- Data becomes more learnable

BATCH NORMALISATION

- Data Preprocessing
- Subtraction of the mean, followed by division with standard division
- Data becomes more learnable



UPDATING WEIGHTS - VANILLA SGD

DONEC QUIS NUNC

UPDATING WEIGHTS - VANILLA SGD

DONEC QUIS NUNC

- Weight = weight - lr*gradient

UPDATING WEIGHTS - VANILLA SGD

DONEC QUIS NUNC

- Weight = weight - lr*gradient
- Issues?

UPDATING WEIGHTS - VANILLA SGD

DONEC QUIS NUNC

- Weight = weight - lr*gradient
- Issues?
- ZIG-ZAG Behavior due to noisy perpendicular mini-batch gradients

UPDATING WEIGHTS - VANILLA SGD

DONEC QUIS NUNC

- Weight = weight - lr*gradient
- Issues?
- ZIG-ZAG Behavior due to noisy perpendicular mini-batch gradients
- Get stuck at Local Minima and Saddle Points. Saddle Points are a bigger problem here

UPDATING WEIGHTS - VANILLA SGD

DONEC QUIS NUNC

- Weight = weight - lr*gradient
- Issues?
- ZIG-ZAG Behavior due to noisy perpendicular mini-batch gradients
- Get stuck at Local Minima and Saddle Points. Saddle Points are a bigger problem here
- Manual Initial Estimation

UPDATING WEIGHTS - VANILLA SGD

DONEC QUIS NUNC

- Weight = weight - lr*gradient
- Issues?
- ZIG-ZAG Behavior due to noisy perpendicular mini-batch gradients
- Get stuck at Local Minima and Saddle Points. Saddle Points are a bigger problem here
- Manual Initial Estimation
- Sensitive learning process as the network converges

SGD + MOMENTUM UPDATE

DONEC QUIS NUNC

SGD + MOMENTUM UPDATE

DONEC QUIS NUNC

- Inspired from a ball rolling on a landscape.

SGD + MOMENTUM UPDATE

DONEC QUIS NUNC

- Inspired from a ball rolling on a landscape.
- Loss is equivalent to the Potential Energy of the ball

SGD + MOMENTUM UPDATE

DONEC QUIS NUNC

- Inspired from a ball rolling on a landscape.
- Loss is equivalent to the Potential Energy of the ball
- Force will be equivalent to our gradient

SGD + MOMENTUM UPDATE

DONEC QUIS NUNC

- Inspired from a ball rolling on a landscape.
- Loss is equivalent to the Potential Energy of the ball
- Force will be equivalent to our gradient
- Hence, the gradient is directly influences by the velocity instead of the position

SGD + MOMENTUM UPDATE

DONEC QUIS NUNC

- Inspired from a ball rolling on a landscape.
- Loss is equivalent to the Potential Energy of the ball
- Force will be equivalent to our gradient
- Hence, the gradient is directly influences by the velocity instead of the position
- v = mu * v - dx; mu => friction

SGD + MOMENTUM UPDATE

DONEC QUIS NUNC

- Inspired from a ball rolling on a landscape.
- Loss is equivalent to the Potential Energy of the ball
- Force will be equivalent to our gradient
- Hence, the gradient is directly influences by the velocity instead of the position
- v = mu * v - dx; mu => friction
- x += lr * v

SGD + MOMENTUM UPDATE

DONEC QUIS NUNC

- Inspired from a ball rolling on a landscape.
- Loss is equivalent to the Potential Energy of the ball
- Force will be equivalent to our gradient
- Hence, the gradient is directly influences by the velocity instead of the position
- v = mu * v - dx; mu => friction
- x += lr * v
- We step in the direction of the velocity vector instead of position.

SGD + MOMENTUM UPDATE

DONEC QUIS NUNC

- Inspired from a ball rolling on a landscape.
- Loss is equivalent to the Potential Energy of the ball
- Force will be equivalent to our gradient
- Hence, the gradient is directly influences by the velocity instead of the position
- v = mu * v - dx; mu => friction
- x += lr * v
- We step in the direction of the velocity vector instead of position.
- No stops at minimas as we update on the basis of velocity. Similar for saddle points. Even though gradient might be zero

ADAGRAD

DONEC QUIS NUNC

ADAGRAD

DONEC QUIS NUNC

```
# Assume the gradient dx and parameter vector x
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

ADAGRAD

DONEC QUIS NUNC

```
# Assume the gradient dx and parameter vector x
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

- Caching the gradient squared. Divide the gradient by sqrt(cache) Similar to scaling the gradient

ADAGRAD

DONEC QUIS NUNC

```
# Assume the gradient dx and parameter vector x
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

- Caching the gradient squared. Divide the gradient by sqrt(cache) Similar to scaling the gradient
- This helps when gradients along one direction are overshooting and along another they are undershooting. The step becomes normalised.

ADAGRAD

DONEC QUIS NUNC

```
# Assume the gradient dx and parameter vector x
cache += dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

- Caching the gradient squared. Divide the gradient by sqrt(cache) Similar to scaling the gradient
- This helps when gradients along one direction are overshooting and along another they are undershooting. The step becomes normalised.
- Problem: What happens as we keep progressing the training using this algorithm?

RMSPROP

DONEC QUIS NUNC

RMSPROP

DONEC QUIS NUNC

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

RMSPROP

DONEC QUIS NUNC

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

- Add a decay rate to the cache

RMSPROP

DONEC QUIS NUNC

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

- Add a decay rate to the cache
- Better optimisation

RMSPROP

DONEC QUIS NUNC

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

- Add a decay rate to the cache
- Better optimisation
- We still face the problem of stopping at Saddle Points

RMSPROP

DONEC QUIS NUNC

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

- Add a decay rate to the cache
- Better optimisation
- We still face the problem of stopping at Saddle Points
- Add Momentum = Adam

ADAM

DONEC QUIS NUNC

ADAM

DONEC QUIS NUNC

```
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
x += - learning_rate * m / (np.sqrt(v) + eps)
```

ADAM

DONEC QUIS NUNC

```
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
x += - learning_rate * m / (np.sqrt(v) + eps)
```

- RMSProp with Momentum

ADAM

DONEC QUIS NUNC

```
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
x += - learning_rate * m / (np.sqrt(v) + eps)
```

- RMSProp with Momentum
- SGD Momentum with Squared Gradients