CMT307 Applied Machine Learning

Session 11

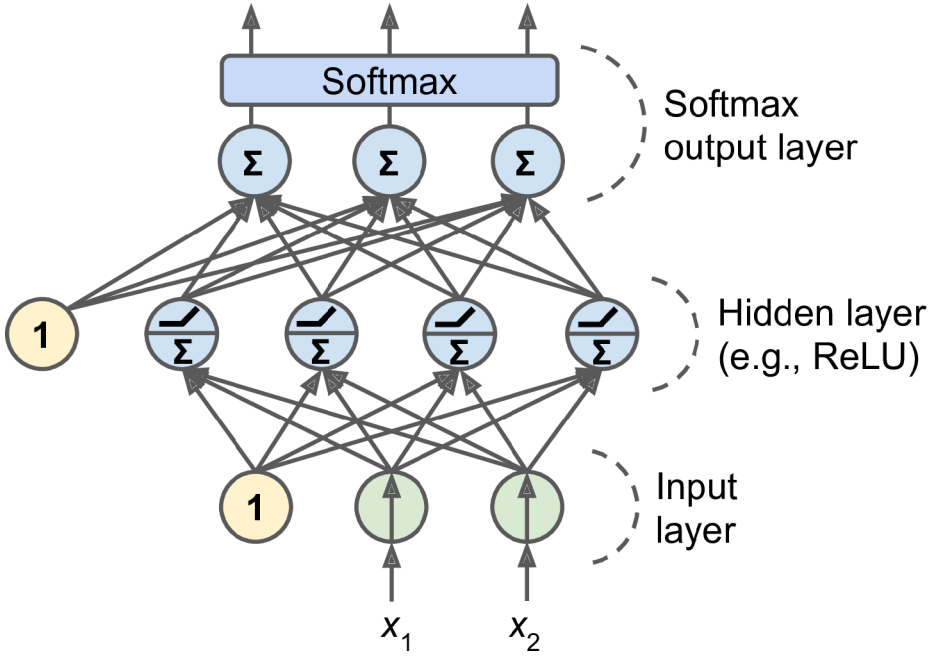
Optimisation, Stochastic Gradient Descent, Loss Functions

Simple Example: Image Classification

- Image Classification with Fashion MNIST
  - Training set: 60,000 images
  - Test set: 10,000 images
  - Each image: 28x28 grayscale
  - Label for 10 classes
  - Drop-in replacement with MNIST – handwriting digit classification
  - More challenging than MNIST

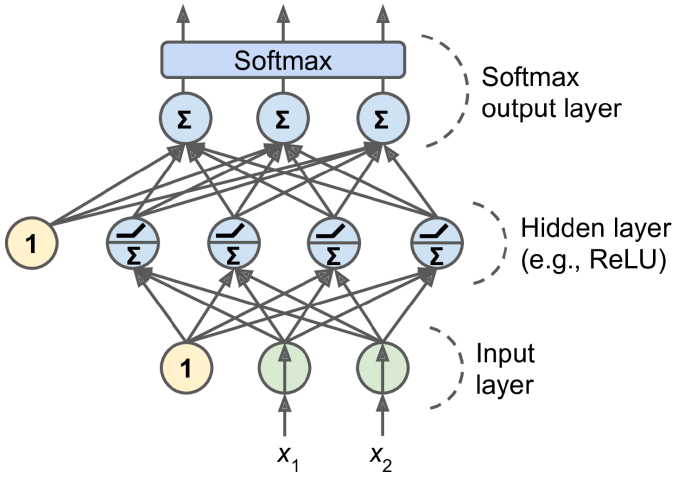
Simple Example: Neural Network

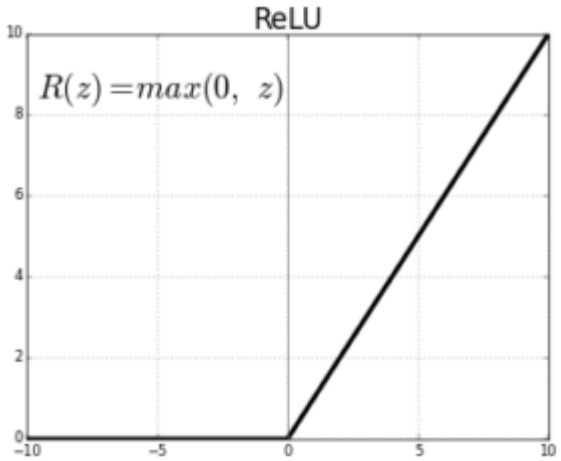
- We use the Keras Sequential model to build the neural network
  - For a sequence of layers
  - Flatten layer: maps a 28x28 image to a long vector; *input_shape* specifies the size of each input sample (28x28 in this case).
  - Dense layers: MLPs with specified number of neurons
  - We use 300, 100 neurons for two hidden layers
  - We use 10 neurons for the output layer, matching the number of categories to predict



Simple Example: Activation

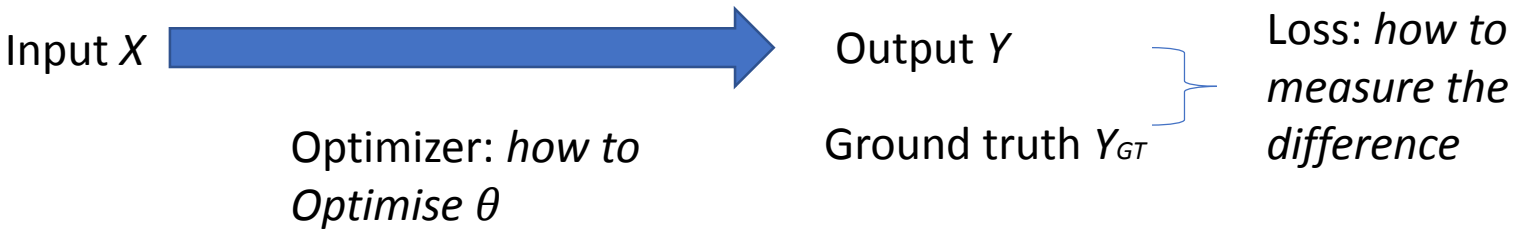
- Activation:
  - This adds non-linear mapping. Without non-linear activation, multilayer perceptions are no more powerful than a single layer!
  - We use ‘relu’ for now (Rectified Linear Unit), except for the last layer
  - We use ‘softmax’ for the output layer:
    - The output can be seen as probabilities that sum to 1
    - often suitable for classification
  - More next week

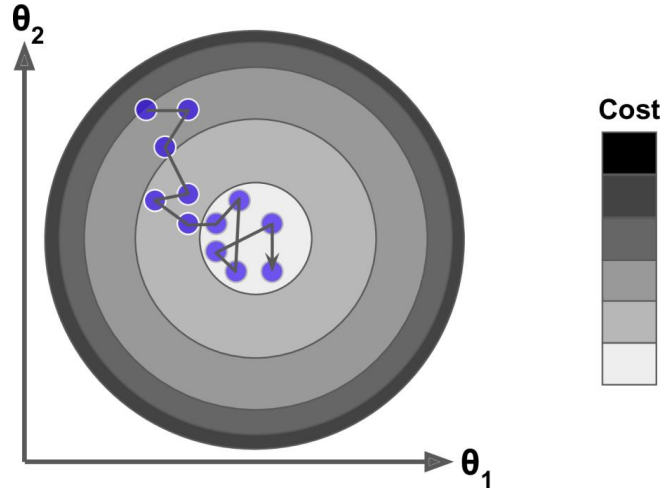




Simple Example: Loss, Optimizer and Metrics

- The following need to be decided when training a neural network:
  - Loss: We use ‘sparse_categorial_crossentroy’ to measure differences of two probability distributions (more coming later today)
  - Optimizer: We use ‘sgd’, meaning Stochastic Gradient Descent (more coming later today)
  - Metrics: We use ‘accuracy’, the proportion of inputs that have been correctly classified, suitable for a classification task
  - A neural network has its weights optimised using the Optimizer, to minimise the Loss. Metrics are the measures of the performance we want to achieve.





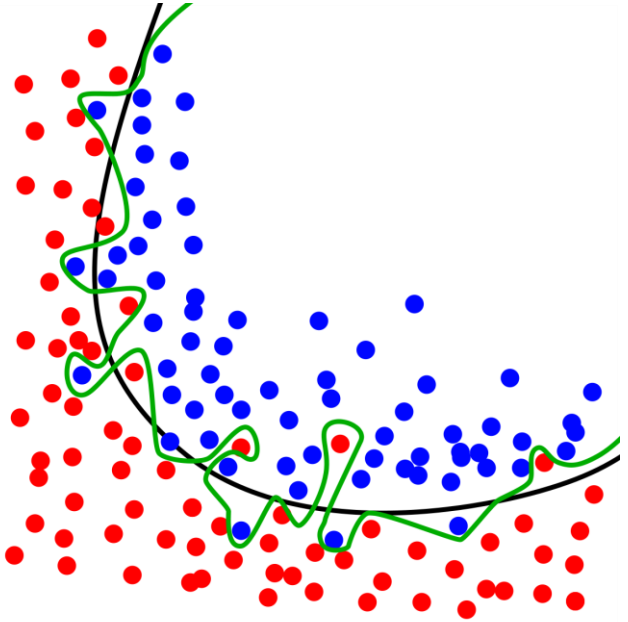
Simple Example: Typical Inputs/Outputs

- Inputs/outputs are problem dependent, for example:
  - This example:
    - Input: an image
    - Output: class label
  - Object detection:
    - Input: an image
    - Output: bounding box (x, y, width, height)
  - House price prediction:
    - Input: house characteristics, e.g. number of rooms, teacher/pupil ratio, distance to major roads, distance to town centres, etc.
    - Output: predicted house value

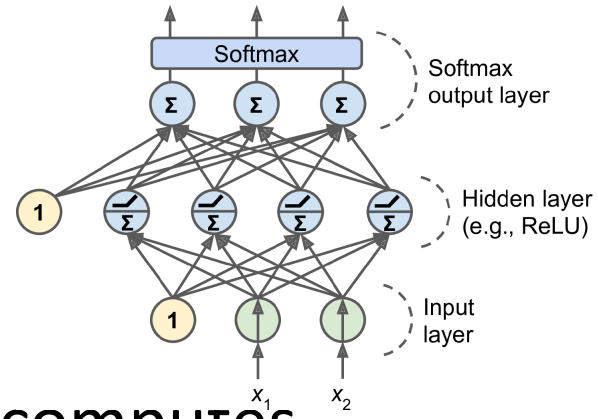
Training / Validation and Test Sets

- Why separating data into training/validation/test sets?
  - Avoid overfitting!
  - Training set: network parameters are optimised towards it
  - Validation set: hyperparameters are optimised towards it
  - Test set: unseen data during training

  - These datasets should NOT have overlaps
  - Often, the performance on the training set > validation set > test set
  - Acceptable if the gap is small
  - If no validation set provided, we may reserve some training data for validation



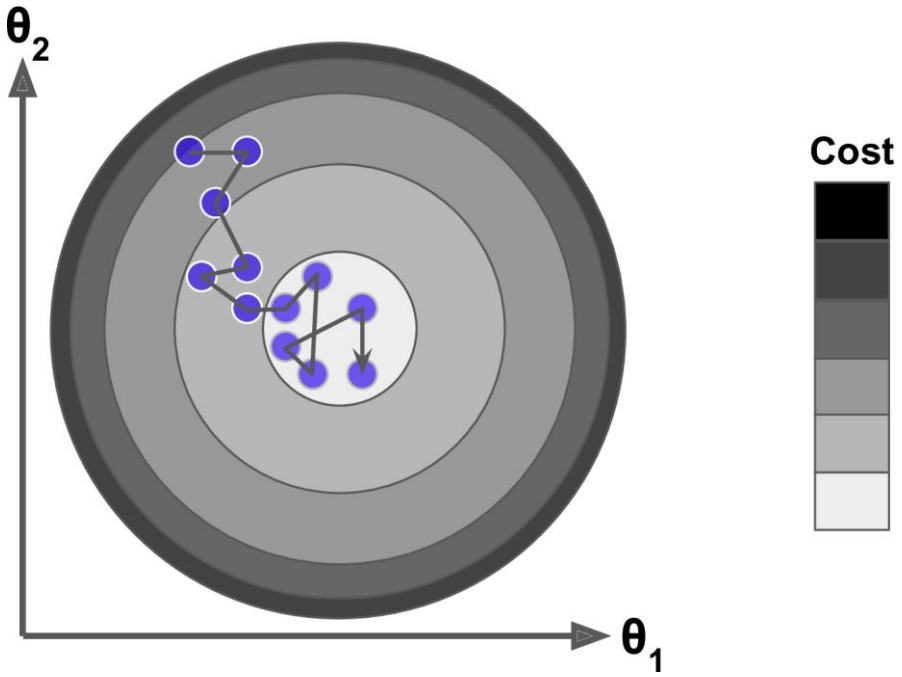
Optimisation: Backpropagation



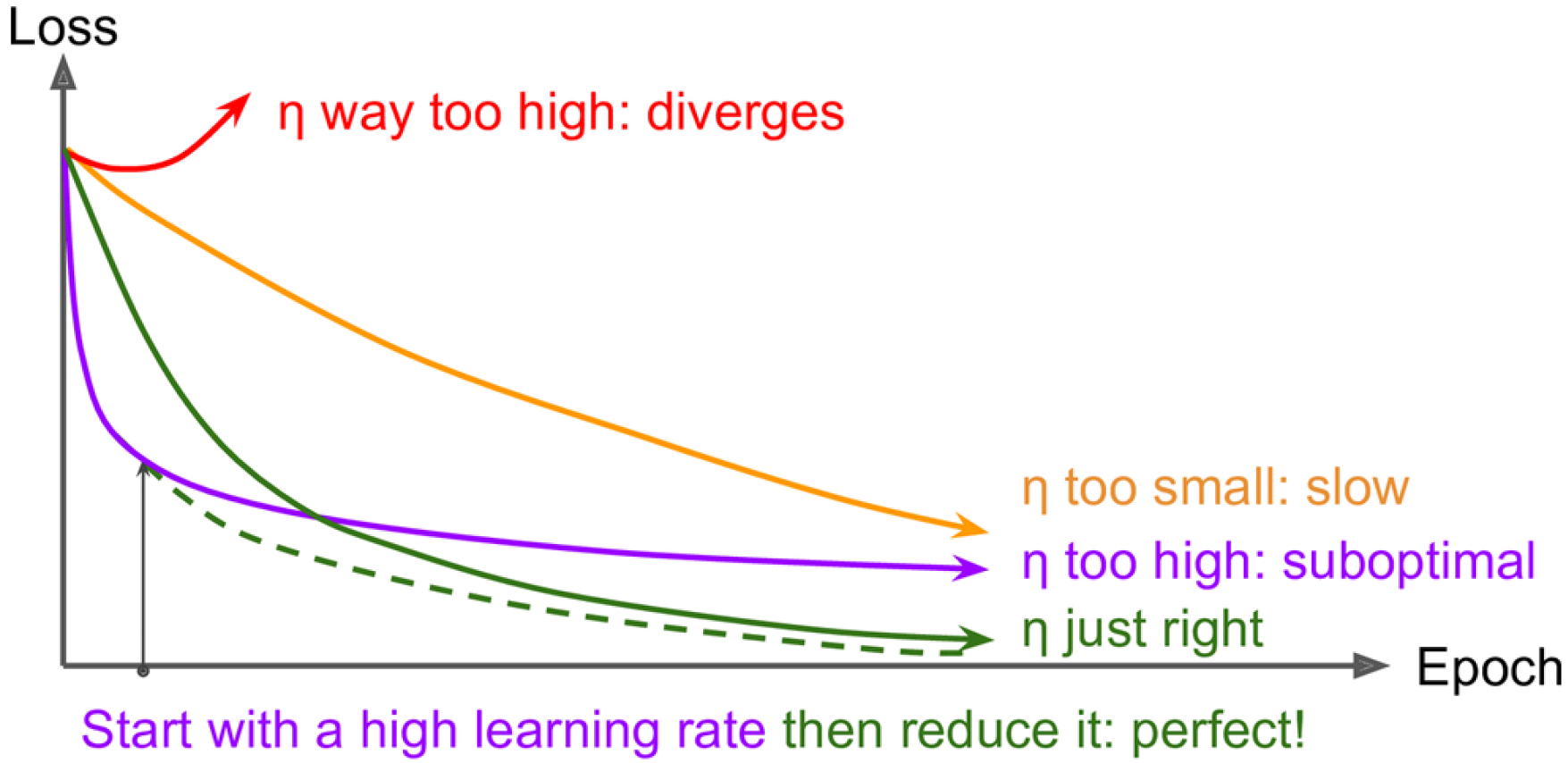
- Proposed in 1986; basis of deep learning
- An efficient algorithm based on *gradient descent* which computes gradients automatically using only *two passes* of the network
  - Take one mini-batch at a time (e.g. with 32 samples)
  - **Forward Pass**: Send the mini-batch to the input of the network, pass through the network and record the outputs of all the neurons
  - Measure the network's output error (using a loss function)
  - Compute the contribution of each output connection to the error
  - **Backward Pass**: Measure how much of the error comes from each connection of each layer, working out backwards from the output layer back to the input layer
  - Gradient Descent: Update the connection weights using gradient descent

Stochastic Gradient Descent (Recap)

- See Session 4 for more details
- Applied to a mini-batch in backpropagation
- Epoch: going through all training data once
- Learning rate:
  - $\theta \leftarrow \theta - \eta \nabla_\theta J(\theta)$
  - Learning rate η: step size in the update
  - η too large: not converging (loss going up)
  - η too small: training too slow
  - Keras: optimizer = keras.optimizers.SGD(lr=**0.01**)
- When to stop: If the performance on the validation set is not improving (early stop), or use the model that gives the best performance on the validation set



Adaptive Learning Rate



Using Keras: optimizer = keras.optimizer.SGD(lr=0.01, decay=1e-4)

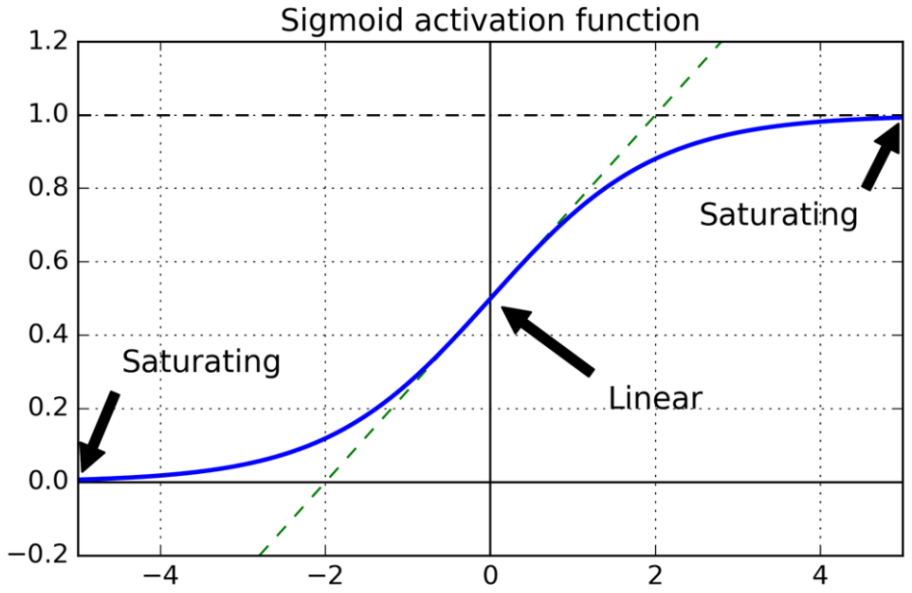
This can significantly improve training efficiency.

Stochastic Gradient Descent: Initialisation

- For SGD to work, the network needs to be initialised, including connection weights (kernel_initializer) and bias (bias_initializer)
  - Bias can be initialized to zeros (bias_initializer = ‘zeros’)
  - However kernel must not be initialized to zeros, as all the neurons in the same layer will be updated in the same way, not providing any extra information!
  - This leads to very poor performance
  - Random initializer normally works fine
  - Better initializers are designed to avoid saturation and improve training

The Vanishing/Exploding Gradients Problems

- Sometimes, gradients may become very small (vanishing) or very large (exploding)
  - The network does not improve much
  - Often due to (locally) neurons are saturated
  - Better initialization that takes into account the input/output connections helps
  - Also: Batch normalization



Batch Normalisation (BN)

- Effective to improve training and avoid vanishing/exploding problems
- During training:
  - Work out the mean and standard deviation of the input in the **mini-batch** to the neuron
  - Linearly scale the input to have standardized input (so avoid saturation)
  - Pass through the neuron
  - Apply the reverse linear scaling to the result
- During testing:
  - The input in the testing does not usually form a mini-batch
  - Use the scaling obtained through training

Batch Normalisation (BN)

- Practical use
  - Effective in practical applications
  - Often converges with fewer epochs, and may improve performance
  - However, each epoch takes longer to train
  - Still worth it in many cases

  - BN can be added
    - After each layer
    - Before or after activation

Gradient Clipping and Gradient Norm Clipping

- To address **exploding gradients**
  - Gradients can be clipped

  optimizer = keras.optimizer.SGD(clipvalue=1.0)

  This clips the partial derivative of every trainable parameter to [-1.0, 1.0]

  But this may change the direction of the gradient [0.1, 10] => [0.1, 1]

  - Gradients can be clipped by norm (length)

  optimizer = keras.optimizer.SGD(clipnorm=1.0)

  If the norm (length) of the gradient is larger than 1.0, it is normalised to length 1.0. So [0.1, 10] => [0.0099995, 0.99995]. The direction is unchanged.

Faster Optimizers

- Default SGD is effective, but could be improved
- Keras provides several optimizer implementation, and can be easily used
  - Momentum Optimisation
  - Adaptive Gradient (AdaGrad)
  - ADAM: Adaptive Moment Estimation

Momentum Optimisation

- Not just move along the negative gradient direction, but also keep the momentum

1. $\mathbf{m} \leftarrow \beta \mathbf{m} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \mathbf{m}$

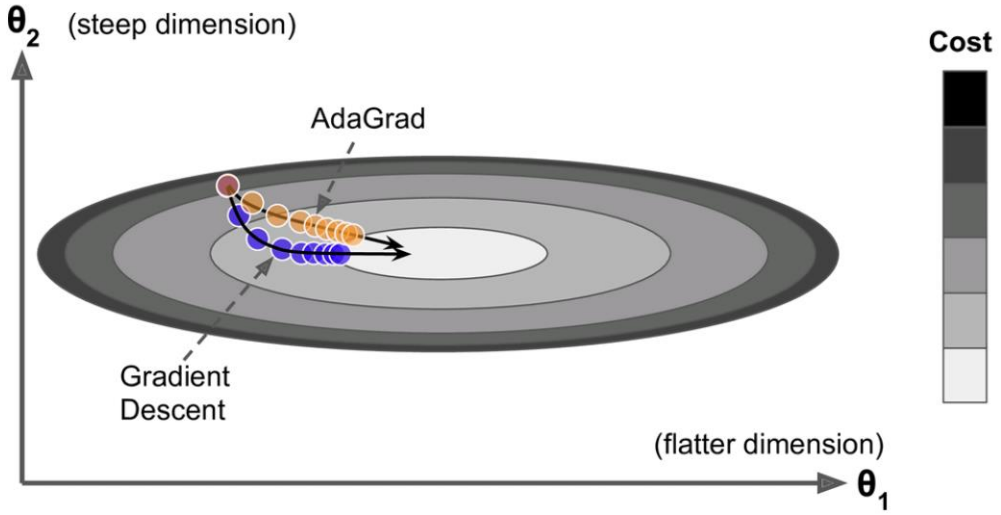
- β: momentum, typical 0.9
- Keras: optimizer = keras.optimizer.SGD(lr=0.001, momentum=0.9)
- Momentum is often useful in practice

AdaGrad

- Not just move along the negative gradient direction, but apply a scaling that is based on accumulated gradients

1. $\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \varepsilon}$

- In practice, suitable for simpler regression problems, but not neural networks.



ADAM: Adaptive Moment Estimation

- Combine the ideas of momentum and adaptive scaling
- Two hyperparameters:
  - Momentum decay β_1, typical 0.9
  - Scaling decay β_2, typical 0.999
- One of the most popular optimizers
- Keras: optimizer = keras.optimizers.Adam(lr=0.001, beta_1=0.9, beta_2= 0.999)

Loss Functions

- Loss functions are the target the network is optimised for during training
  - Generally problem specific
  - Typical loss functions are commonly used, and provided by Keras
    - Regression problems: Mean Squared Error (MSE), Mean Absolution Error (MAE)
    - Classification problems: Cross Entropy
  - It is also possible to define custom loss functions

Loss Functions: MSE & MAE

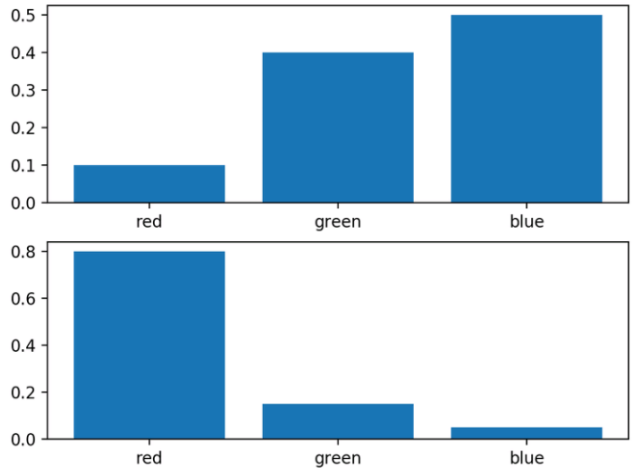
- Usually for Regression Problems
- $MSE = \frac{1}{N}\sum_{i=1}^{N}\left(Y_i - \hat{Y}_i\right)^2$ (mean_squared_error in Keras)
- $MAE = \frac{1}{N}\sum_{i=1}^{N}\left|Y_i - \hat{Y}_i\right|$ (mean_absolute_error in Keras)

- MSE: penalise larger errors more
- MAE: more robust to outliers

Loss Function: Cross Entropy

One hot vector: 3 =>
[0, 0, 0, 1, 0, 0…]

- Classification results can be seen as a probability distribution
- The ground truth label can be treated as a one-hot vector
- This becomes the problem of comparing two distributions
- Entropy: from Information Theory, the number of bits (amount of information) needed to transmit the information
  - Skewed distribution: low entropy (unsurprising)
  - Balanced distribution: high entropy (surprising)
- Cross-entropy: the number of bits needed for representing one source when coding is optimised towards the other
- Essentially measures how similar are the distributions



Loss Function: Cross Entropy

- Cross entropy is useful for classification, Keras supports:
  - **sparse_categorical_crossentropy**: when the ground truth is the label, rather than one-hot vector (often use ‘softmax’ as the activation for the output layer)
  - **categorical_crossentropy**: when the ground truth is represented as one-hot vector (often use ‘softmax’ as the activation for the output layer)
  - **binary_crossentropy:** for binary (2-class) cases

- Time for Practice!