Training Neural Networks

Learning Objectives

After completing this lecture, you will be able to:-

- Apply appropriate batching when using gradient descent for training neural networks
- Implement neural networks with Keras
- Use neural networks in multiclass problems using one-hot-encoding or similar techniques
- Describe various techniques for regularization of neural network training

Decisions for Training

- Given a group of examples, we know how to compute the derivative for each weight
- How exactly do we update the weights?
- How often do we update the weights?
  - After each data point?
  - After all training data points?
- Choices made here could make training faster / better / etc. (or the reverse)

Batch Gradient Descent

- Classical approach – get derivative for entire data set, then take a step in that direction

$$W_{new} = W_{old} - \alpha \frac{\delta J}{\delta W}$$

- Pros: Each step is informed by all the data
- Cons: Very slow, especially as data gets big

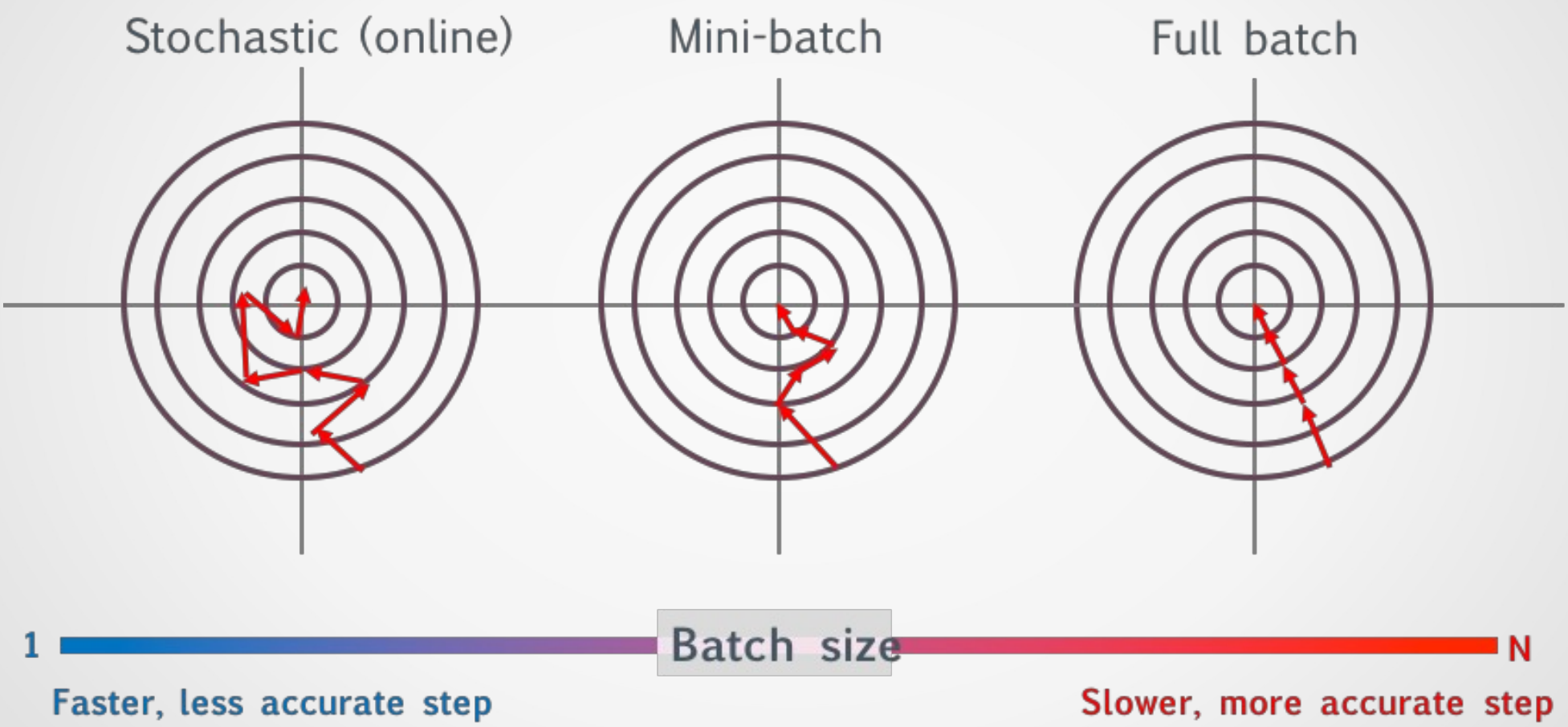
Stochastic Gradient Descent

- Get derivative for just one point, take a step in that direction
- Steps are “less informed” but you take more of them (per computation time)
- Should “balance out”
- Probably want a smaller step size
- Also helps to regularize

Compromise: Mini-batch

- Get derivative for a ‘small’ set of points, then take a step in that direction
- Typical mini batch sizes are 16, 32
- Strikes a balance between two extremes

Batching Approaches Comparison



Batching Terminology

- Full-batch
  - Use entire data set to compute gradient before updating
- Mini-batch
  - Use a smaller portion of data (more than one) to computer gradient before upgrading
- Stochastic Gradient Descent (SGD)
  - Use a single example to compute gradient before updating (sometimes just refers to mini-batch)

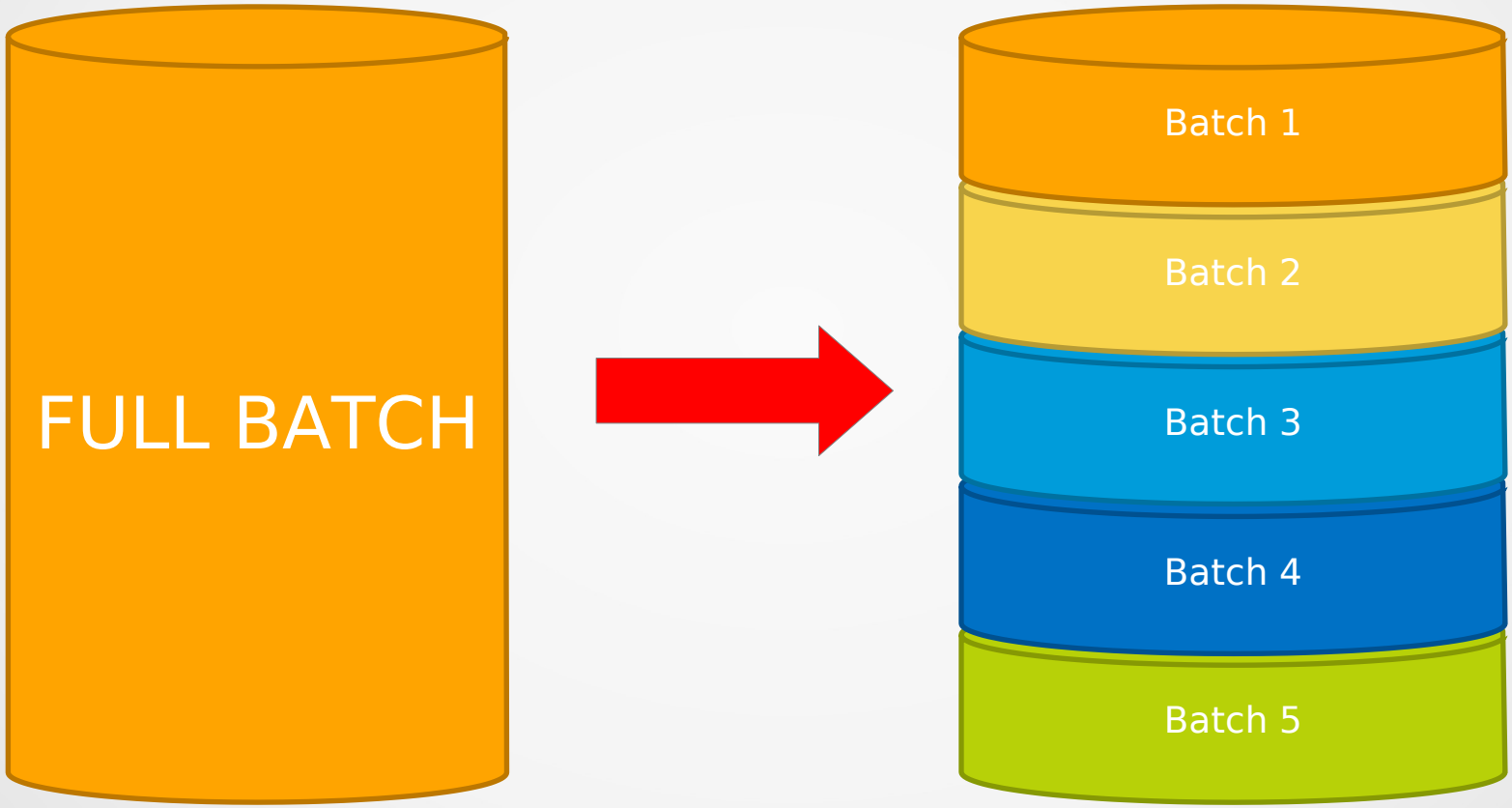
Batching Terminology

- An **epoch** refers to a single pass through all the training data
- In full batch gradient descent, there would be one step taken per epoch
- In SGD / Online learning, there would be *n* steps taken per epoch (*n* is the size of the training set)
- In Mini-batch there would be (*n*/batch size) steps taken per epoch
- When training, it is common to refer to the number of epochs needed for the model to be “trained”

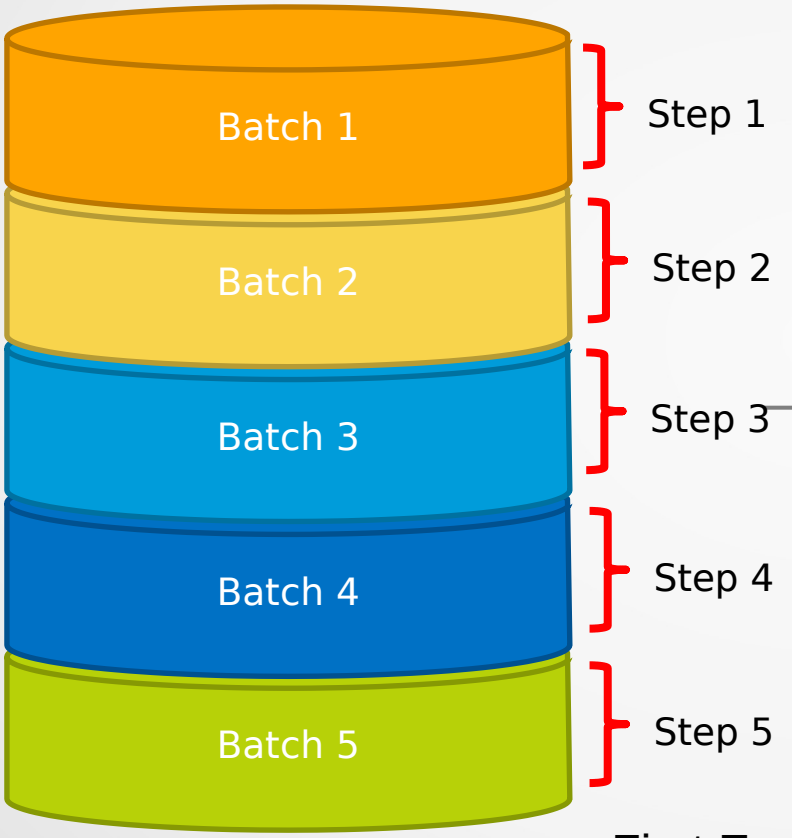
Batching Terminology

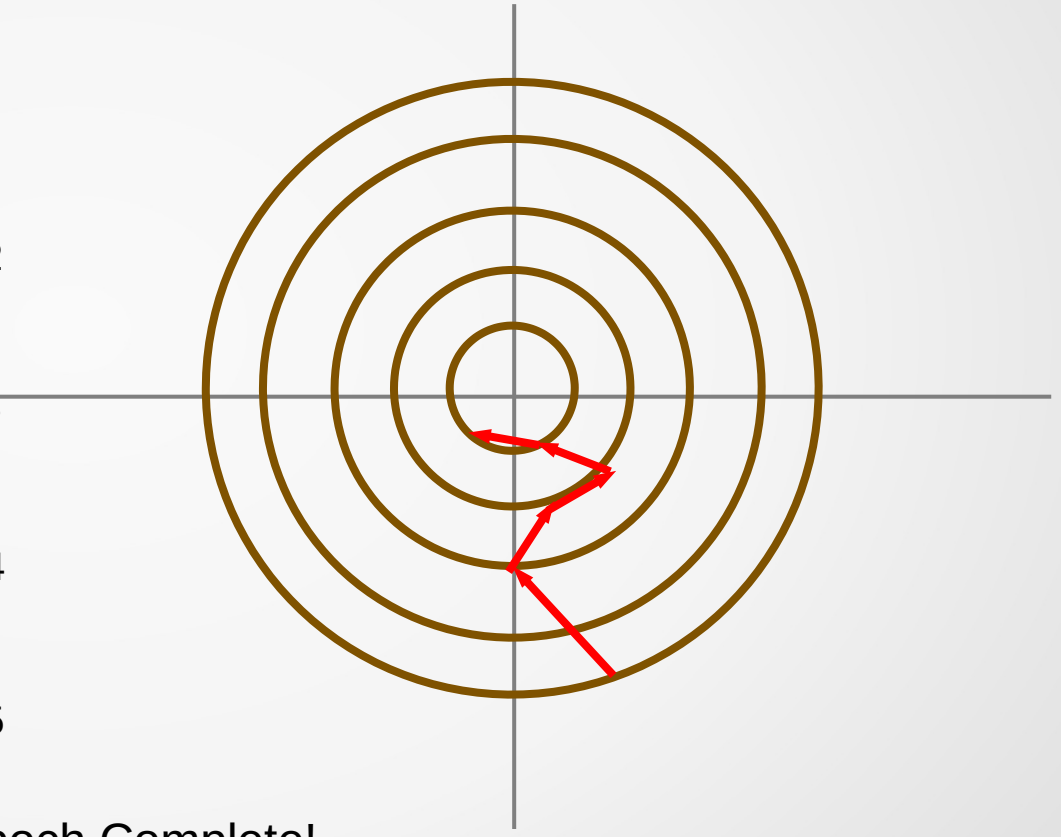
- It is common to implement **data shuffling**
  - Cyclical movement is possible if the same data is used for mini-batch/SGD in the same order
  - Shuffling the data after each epoch can avoid this, and aid convergence
  - This way, the data is not ‘seen’ by the network in the same order every time
  - Also, each set of batches is different from any of the sets used in the previous epoch

Batching Example



Batching Example

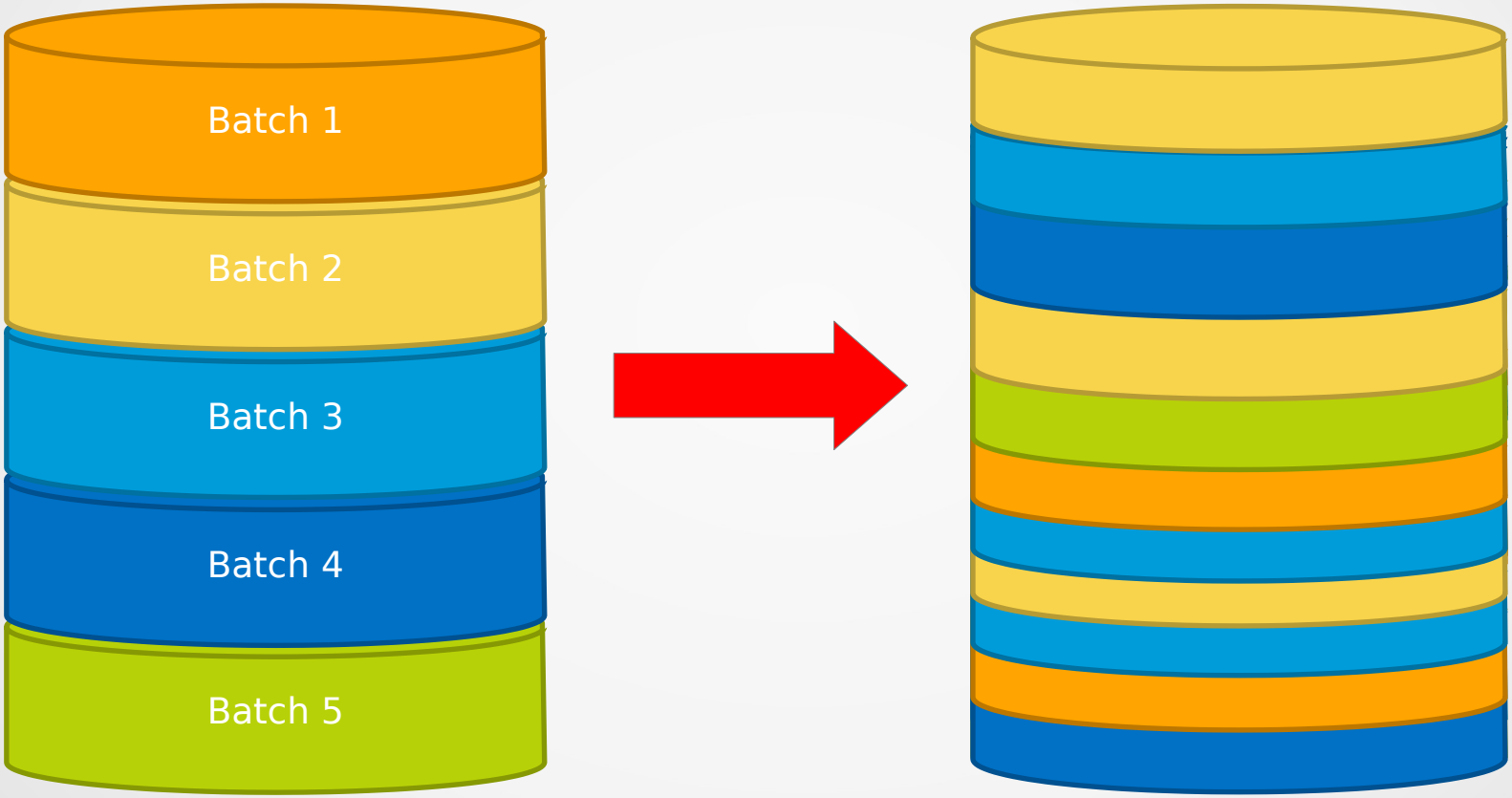




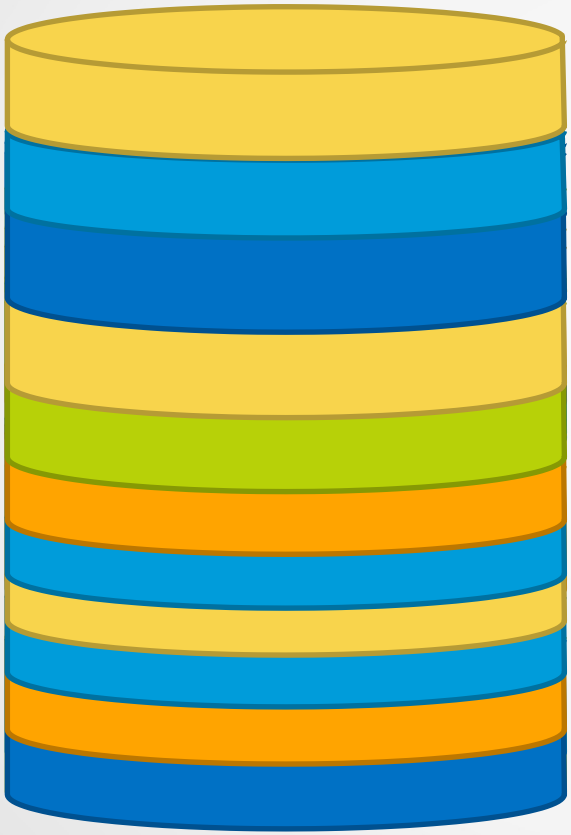
First Epoch Complete!

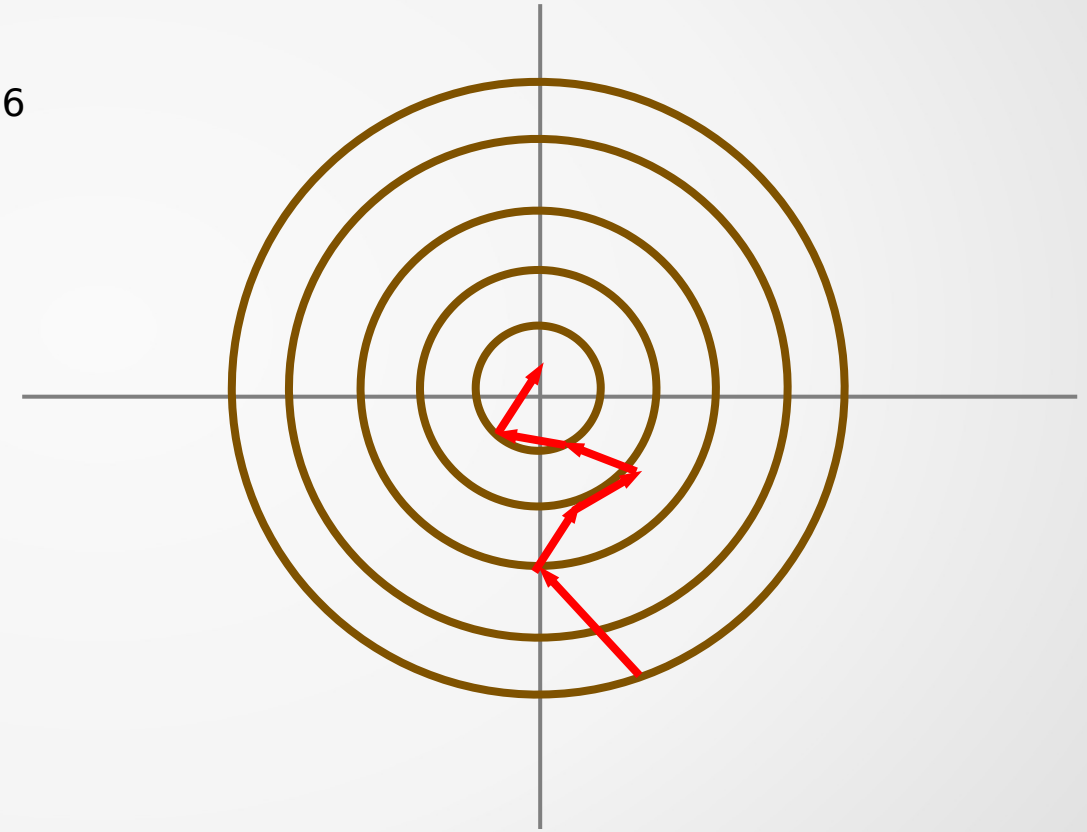
Batching Example

Shuffle the Data!



Batching Example





The Keras Package

- Keras allows easy construction, training, and execution of Deep Neural Networks
- Written in Python, and allows users to configure complicated models directly in Python
- Uses either Tensorflow or Theano “under the hood”
- Uses either CPU or GPU for computation
- Uses numpy data structures, and a similar command structure to scikit-learn (`model.fit`, `model.predict`, etc.)

Typical Command Structure in Keras

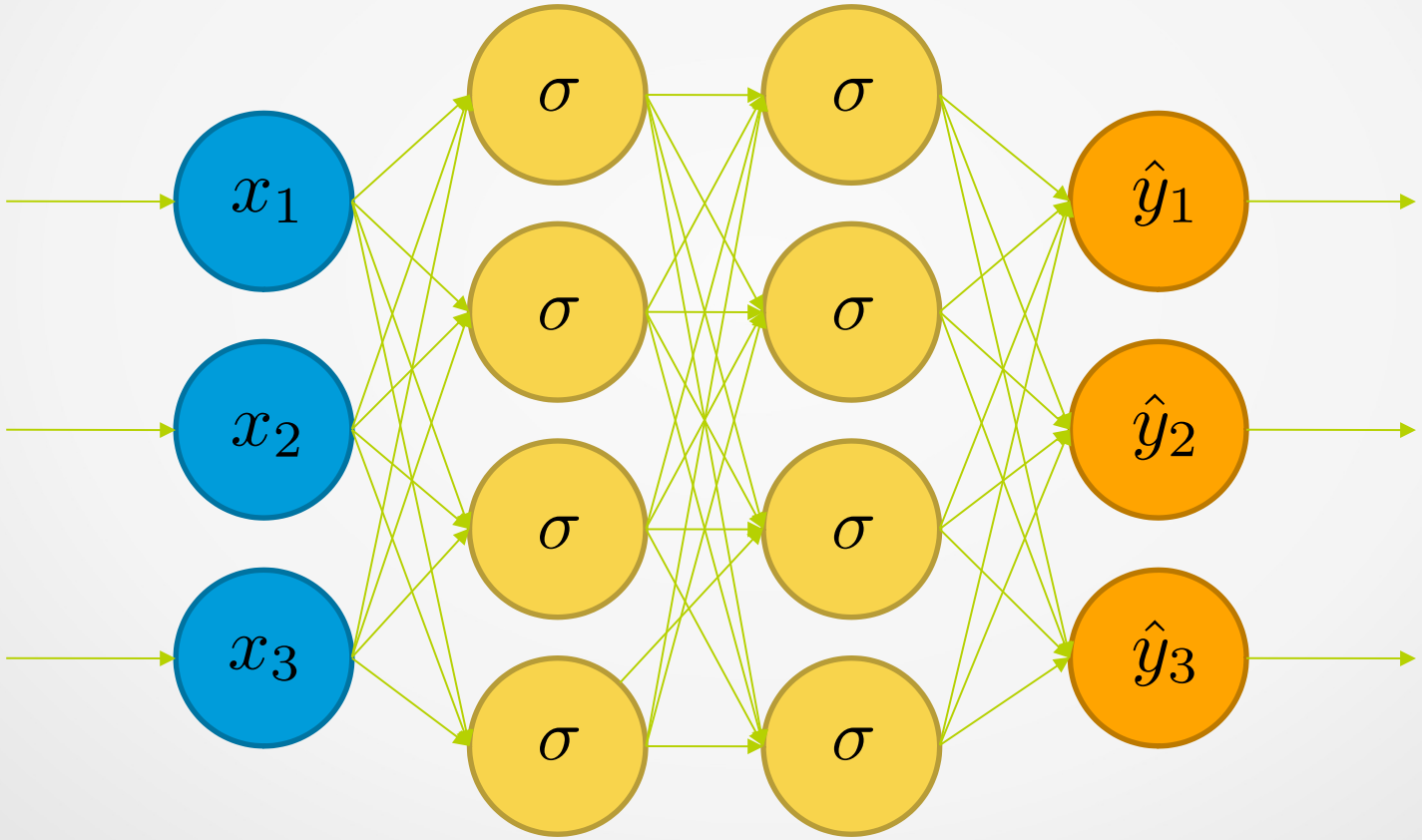
- Build the structure of your network
- Compile the model, specifying your loss function, metrics, and optimizer (which includes the learning rate)
- Fit the model on your training data (specifying batch size, number of epochs)
- Predict on new data
- Evaluate your results

Building the Model

- Keras provides two approaches to building the structure of your model:
- Sequential Model: allows a linear stack of layers – simpler and more convenient if model has this form
- Functional API: more detailed and complex, but allows more complicated architectures
- We will focus on the Sequential Model

Keras Sequential Model Example

Let's build this Neural Network structure in Keras



Keras Sequential Model Example

First, import the Sequential function and initialize your model object

```
from keras.models import Sequential
Model = Sequential()
```

Keras Sequential Model Example

Then we add the layers to the model one by one

```
from keras.layers import Dense, Activation

# For the first layer, specify the input dimension
model.add(Dense(units=4, input_dim=3))

# Specify an activation function
model.add(Activation('sigmoid'))

# For subsequent layers, the input dimension is
# presumed from the previous layer
model.add(Dense(units=4))
model.add(Activation('sigmoid'))
model.add(Dense(units=3))
model.add(Activation('softmax'))
```

Multiclass Neural Network Classification

- For binary classification problems, we have a final layer with a single node and a sigmoid activation function
- This has many desirable properties
  - Output is strictly between 0 and 1
  - Can be interpreted as a probability
  - Derivative is ‘nice’
  - Analogous to logistic regression
- Is there a natural way to extend this to a multiclass setting?

Multiclass Neural Network Classification

- Reminder: one hot encoding for categories
- Take a vector with length equal to the number of categories
- Represent each category with one at a particular position (and zero everywhere else)

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Cat Dog Toaster

Multiclass Neural Network Classification

- For multiclass classification problems, let the final layer be a vector with length equal to the number of possible classes
- Extension of sigmoid to multiclass is the ‘softmax’ function

$$softmax(z_i) = \frac{e^{Z_i}}{\sum_{k=1}^{K} e^{Z_k}}$$

- Yields a vector with entries that are between 0 and 1, and sum to 1

Multiclass Neural Network Classification

- For loss function use “categorical cross entropy”
- This is just the log-loss function in disguise

$$C.E = -\sum_{i=1}^{n} y_i \log(\hat{y}_i)$$

- Derivative has a nice property when used with softmax

$$\frac{\delta(C.E.)}{\delta(softmax)} \cdot \frac{\delta(softmax)}{\delta z_i} = \hat{y}_i - y_i$$

Reminder: Scaling

- Linear scaling to the interval [0, 1]

$$x_i = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

- Linear scaling to the interval [-1, 1]

$$x_i = 2\left(\frac{x_i - \bar{x}}{x_{max} - x_{min}}\right) - 1$$

- Standardization (making variable approximately standard normal)

$$x_i = \frac{x_i - \hat{x}}{\sigma} \qquad \sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{x})^2}$$

Regularizing Neural Networks

We have several means by which to help “regularize” neural networks – that is, to prevent overfitting

- Regularization penalty in cost function
- Dropout
- Early stopping
- Stochastic / Mini-batch Gradient descent (to some degree)

Regularizing Neural Networks

Penalized Cost Function

- One option is to explicitly add a penalty to the loss function for having high weights.
- This is a similar approach to Ridge Regression

$$J = \frac{1}{2n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 + \lambda\sum_{j=1}^{m}W_j^2$$

- Can have an analogous expression for Categorical Cross Entropy

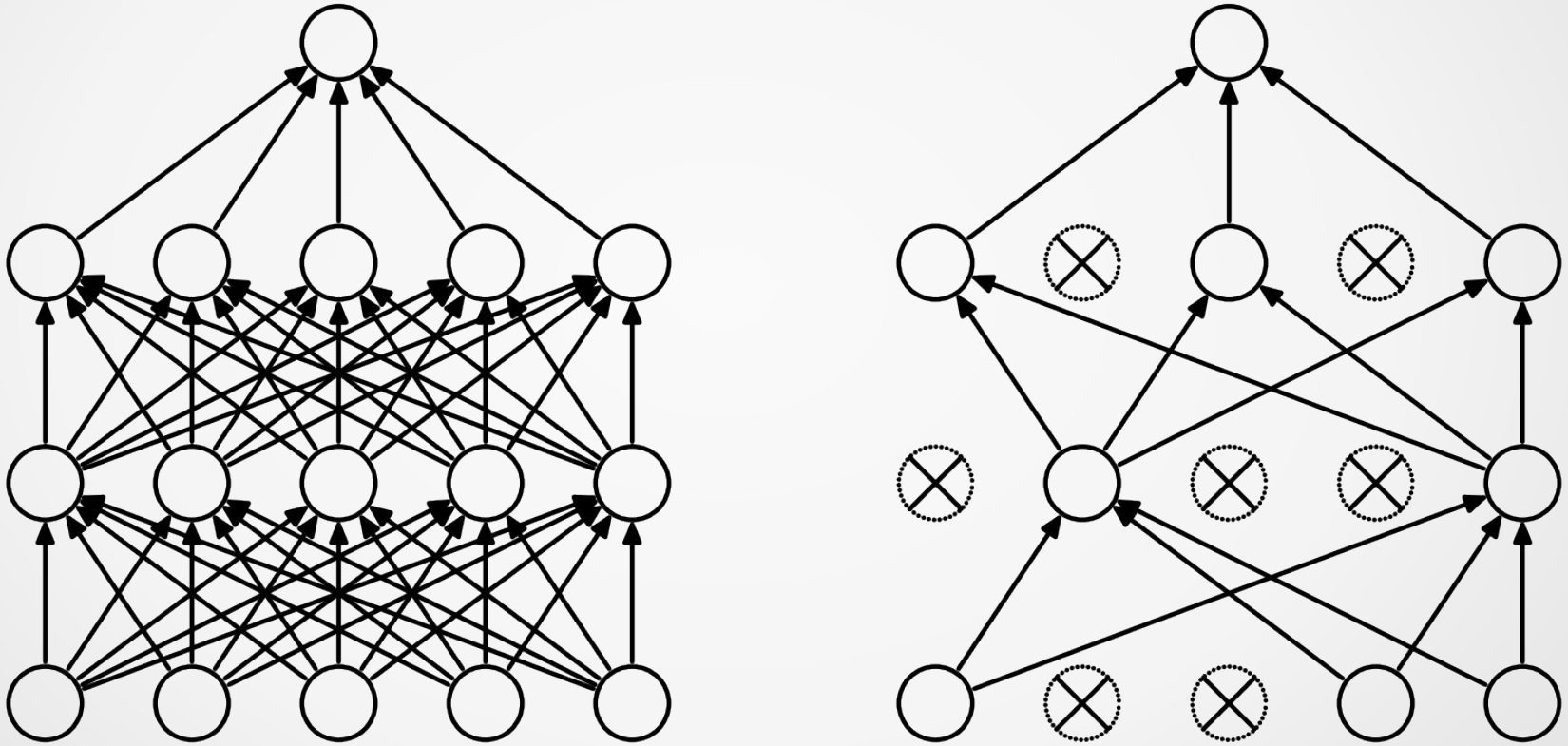
Regularizing Neural Networks

Dropout

- Dropout is a mechanism where at each training iteration (batch) we randomly remove a subset of neurons
- This prevents the neural network from relying too much on individual pathways, making it more “robust”
- At test time we “rescale” the weight of the neuron to reflect the percentage of the time it was active

Regularizing Neural Networks

Dropout Visualization



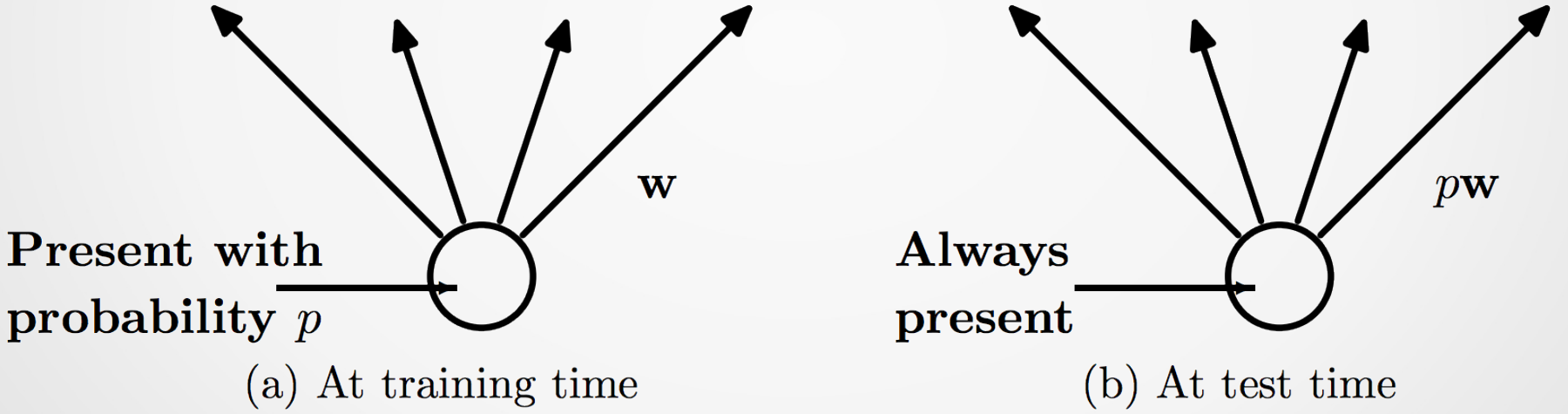
(a) Standard Neural Net

(b) After applying dropout.

Regularizing Neural Networks

Dropout Visualization

- If the neuron was present with probability *p*, at test time we scale the outbound weights by a factor of *p*.



(a) At training time (b) At test time

Regularizing Neural Networks

Early Stopping

- Another more heuristical approach to regularization is early stopping
- This refers to choosing some rules after which to stop training
- E.g.
  - Check the validation log-loss every 10 epochs
  - If it is higher than it was last time, stop and use the previous model (i.e. from 10 epochs ago)

Regularizing NN – Optimizers

- We have considered approaches to gradient descent which vary the number of data points involved in a step.
- However, they have all used the standard update formula:

$$W := W - \alpha \cdot \nabla J$$

- There are several variants to updating the weights which give better performance in practice.
- These successive “tweaks” each attempt to improve on the previous idea.
- The resulting (often complicated) methods are referred to as “optimizers”.

Regularizing NN – Optimizers

Momentum

- Idea: only change direction by a little bit each time.
- Keeps a “running average” of the step directions, smoothing out the variation of the individual points

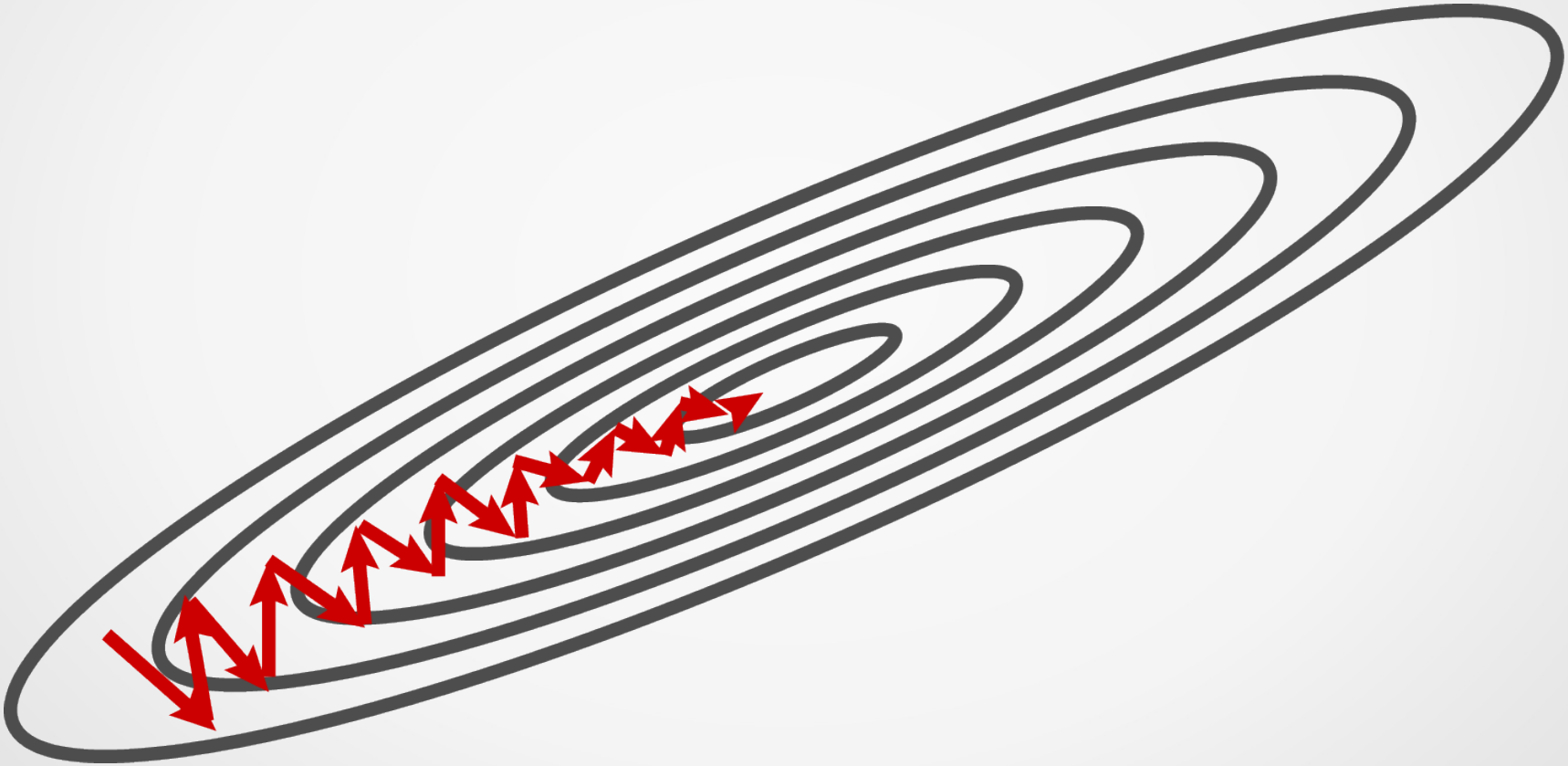
$$v_t := \eta \cdot v_{t-1} - \alpha \cdot \nabla J$$

$$W := W - v_t$$

- Here, is referred to as the “momentum”. It is generally given a value < 1

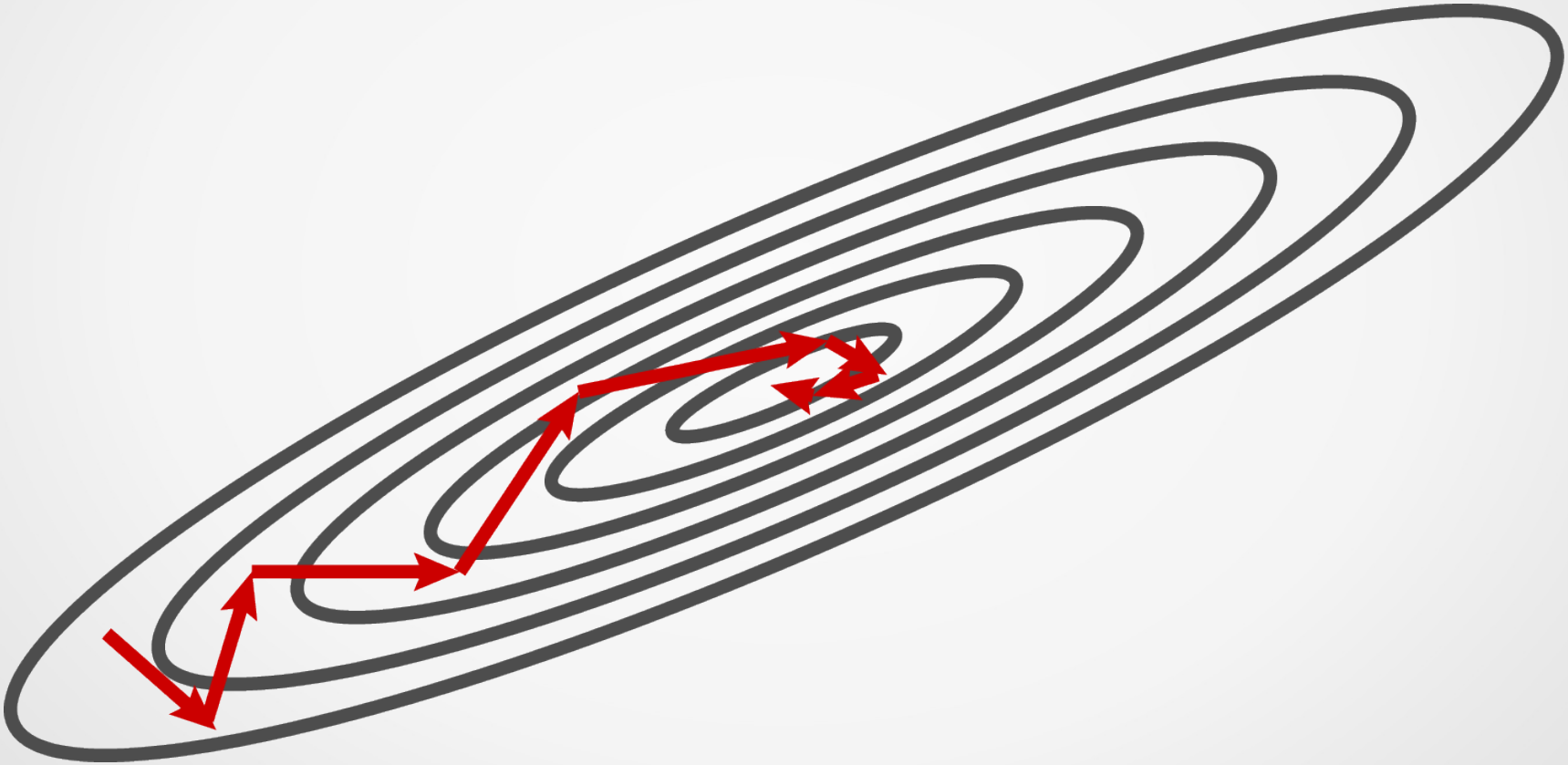
Regularizing NN – Optimizers

Gradient Descent vs Momentum



Regularizing NN – Optimizers

Gradient Descent vs Momentum



Regularizing NN – Optimizers

Nesterov Momentum

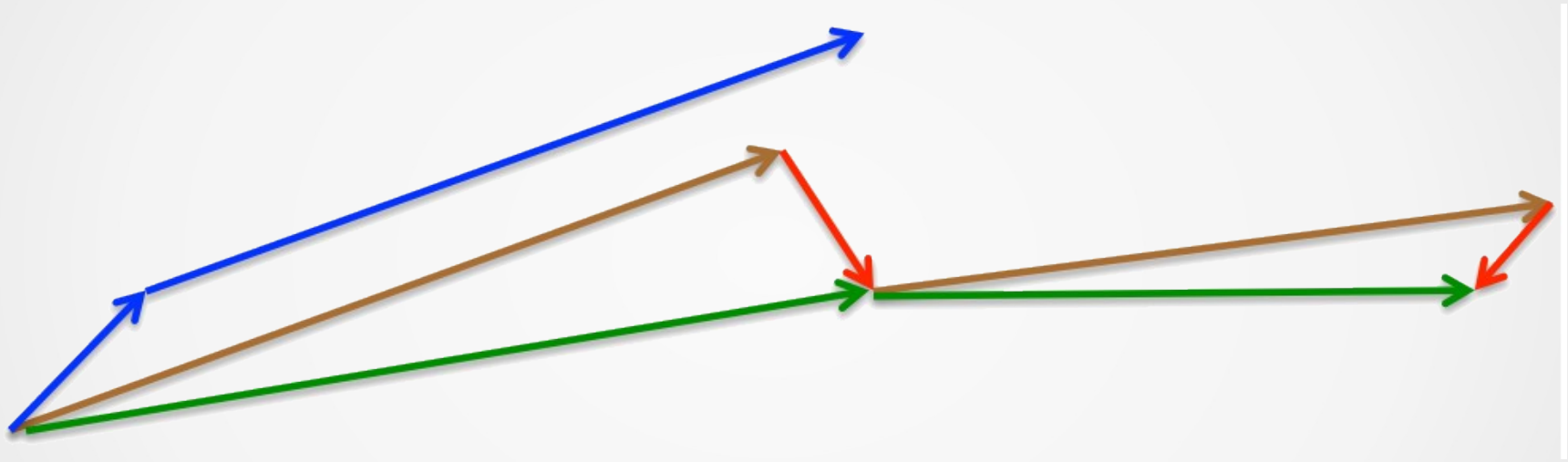
- Idea: control "overshooting" by looking ahead
- Apply gradient only to the "non-momentum" component

$$v_t := \eta \cdot v_{t-1} - \alpha \cdot \nabla(J - \eta \cdot v_{t-1})$$

$$W := W - v_t$$

Regularizing NN – Optimizers

Nesterov Momentum Visualized



Regularizing NN – Optimizers

AdaGrad

- Idea: scale the update for each weight separately
- Update frequently-updated weights less
- Keep running sum of previous updates
- Divide new updates by factor of previous sum

$$W := W - \frac{\eta}{\sqrt{G_t} + \varepsilon} \circ \nabla J$$

Regularizing NN – Optimizers

RMSProp

- Quite similar to AdaGrad
- Rather than using the sum of previous gradients, decay older gradients more than more recent ones
- More adaptive to recent updates

Regularizing NN – Optimizers

Adam

- Idea: use both first-order and second-order change information and decay both over time

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\nabla J \qquad m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla J$$

$$\hat{v}_t = \frac{v_t}{1-\beta_1^t} \qquad \hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$W := W - \frac{\eta}{\sqrt{\hat{v}_t} + \varepsilon} \circ \hat{m}_t$$

Regularizing NN – Optimizers

- RMSProp and Adam seem to be quite popular now.
- Difficult to predict in advance which will be best for a particular problem.
- Still an active area of inquiry.

End of Lecture

Many thanks to Intel Software for providing a variety of resources for this lecture series