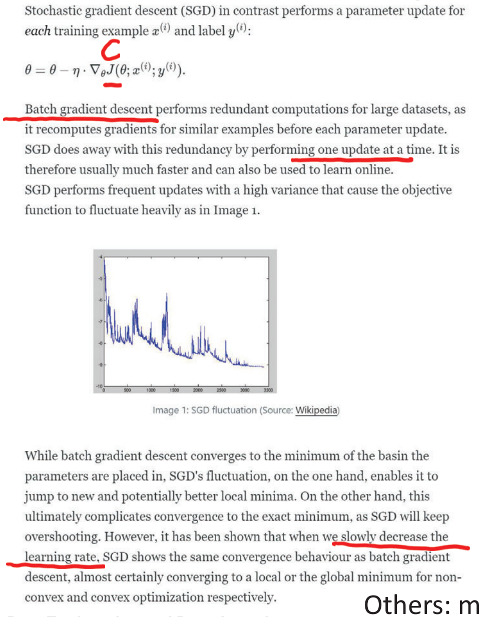
Optimizers

= Batchsize 1

Stochastic Gradient Descent (SGD)



Stochastic gradient descent (SGD) in contrast performs a parameter update for *each* training example $x^{(i)}$ and label $y^{(i)}$:

C

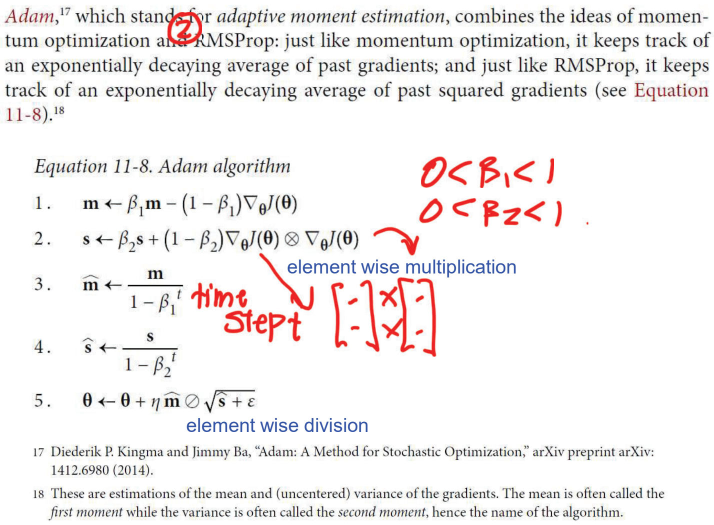
$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}).$$

Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and can also be used to learn online.
SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily as in Image 1.

Image 1: SGD fluctuation (Source: Wikipedia)

While batch gradient descent converges to the minimum of the basin the parameters are placed in, SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima. On the other hand, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting. However, it has been shown that when we slowly decrease the learning rate, SGD shows the same convergence behaviour as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively.

Adaptive Moment (Adam)



Adam,[17] which stands for *adaptive moment estimation*, combines the ideas of momentum ① optimization and ② RMSProp: just like momentum optimization, it keeps track of an exponentially decaying average of past gradients; and just like RMSProp, it keeps track of an exponentially decaying average of past squared gradients (see Equation 11-8).[18]

Equation 11-8. Adam algorithm

$0<\beta_1<1$

$0<\beta_2<1$

1. $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1-\beta_1)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2. $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1-\beta_2)\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$

element wise multiplication

$\begin{bmatrix}\cdot\\\cdot\end{bmatrix}\times\begin{bmatrix}\cdot\\\cdot\end{bmatrix}$

3. $\widehat{\mathbf{m}} \leftarrow \dfrac{\mathbf{m}}{1-\beta_1{}^t}$ time stept
4. $\widehat{\mathbf{s}} \leftarrow \dfrac{\mathbf{s}}{1-\beta_2{}^t}$
5. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta\, \widehat{\mathbf{m}} \oslash \sqrt{\widehat{\mathbf{s}} + \varepsilon}$

element wise division

[17] Diederik P. Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization," arXiv preprint arXiv: 1412.6980 (2014).

[18] These are estimations of the mean and (uncentered) variance of the gradients. The mean is often called the *first moment* while the variance is often called the *second moment*, hence the name of the algorithm.

source: Géron, Hands-on ML

Others: momentum, RMSprop, Adadelta, Adamax etc.

Hyperparameters

- A **hyperparameter** is a parameter whose value is used to control the learning process (it is different from "weights" or "parameters")
- e.g. the batch size is a hyperparameter of gradient descent that controls the number of training samples to work through before the model's internal parameters or weights are updated
- e.g. the number of epochs is a hyperparameter that controls the number of complete passes through the training dataset
- Other hyperparameters: optimizer (e.g. SGD, Adam) and its parameter settings, no. of layers and neurons in each layer, activation functions etc.

Training, Validation and Test sets

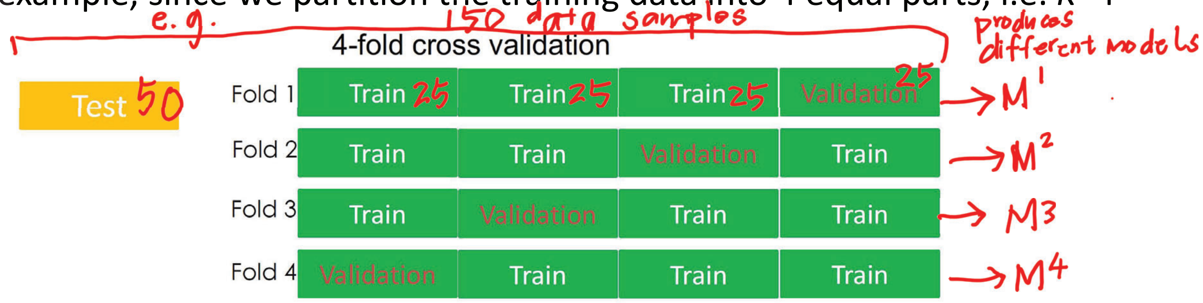
- **<u>Basic method</u>**
- Split entire dataset into **training set** and **validation sets** (we may also wish to set aside a **test set**)
  - The training set is used for training
  - The validation set (fixed) is to **simulate the unseen test data** for **model tuning** during the **training stage**
- Use validation set to decide when to stop training and prevent overfitting

- also used for hyperparameter tuning (see later)

- *Strictly speaking, the **test set** is untouched until the final evaluation.* *<u>note</u>: sometimes, the "test set" is used as the validation set*
- Performance metrics such as mean-squared error, classification accuracy etc.

Training, Validation and Test sets

- **<u>Advanced method with *K*-fold Cross Validation</u>**
- Split entire dataset into **training set** and **test set**
- The **training set** is used for training and validation
- *The **test set** is untouched until the final evaluation*
- Partition the training set and select one partition to be the validation set
  - next, the validation set is rotated until all the training data have been used for validation
  - in this example, since we partition the training data into 4 equal parts, i.e. *K*=4



e.g. 150 data samples

4-fold cross validation

Test 50

Fold 1	Train 25	Train 25	Train 25	Validation 25	→ M^1
Fold 2	Train	Train	Validation	Train	→ M^2
Fold 3	Train	Validation	Train	Train	→ M3
Fold 4	Validation	Train	Train	Train	→ M^4

produces different models

Hyperparameter Optimization / Tuning

- The validation set is used for tuning the hyperparameters during the training stage
  - evaluate the model with different hyperparameters on the validation set

Which hyperparameter value(s) should we use?

e.g.
→ 2 hidden nodes

5 hidden nodes

	Acc. on Val Set 1 **Fold 1**	Acc. on Val Set 2 **Fold 2**	Acc. on Val Set 3 **Fold 3**	Acc. on Val Set 4 **Fold 4**	Avg. Acc on Val. Set
Classifier with hyper-Param1	88% M_1^1	89% M_1^2	93% M_1^3	92% M_1^4	90.5%
Classifier with hyper-Param2	90% M_2^1	88% M_2^2	91% M_2^3	91% M_2^4	90%

✓ ←

- Hence, select <u>hyperparameter 1</u> and retrain using the entire training set to obtain the final model, which can then be evaluated on the test set
- Note: Hyperparameter optimization will be covered in greater detail later

Performance Metrics: Regression

Mean Square Error

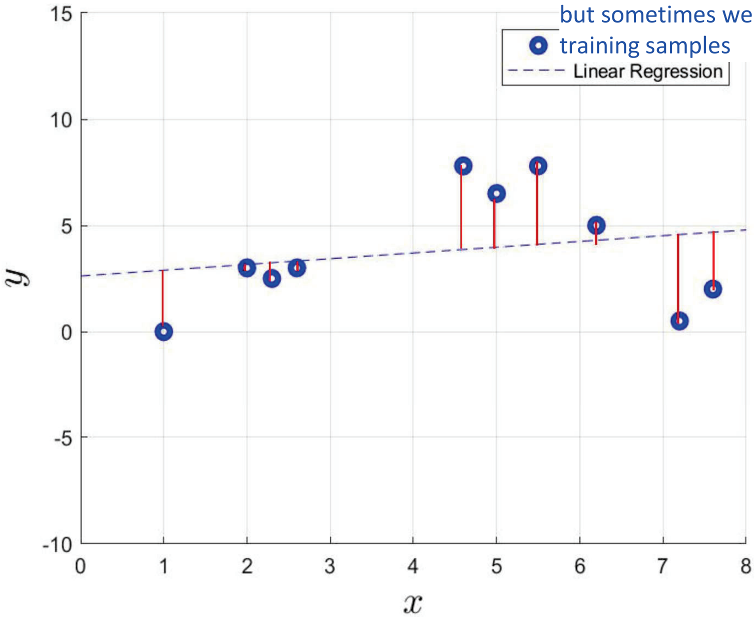
$$(\text{MSE} = \frac{\Sigma_{i=1}^{n}(y_i - \hat{y}_i)^2}{n})$$

Mean Absolute Error

$$(\text{MAE} = \frac{\Sigma_{i=1}^{n}|y_i - \hat{y}_i|}{n})$$

where y_i denotes the target output and $\hat{y}_i$ denotes the predicted output for sample i.

Evaluation is usually done with validation or test samples, but sometimes we evaluate for training samples



Refer to sklearn.metrics and Keras metrics

Performance Metrics: Binary Classification

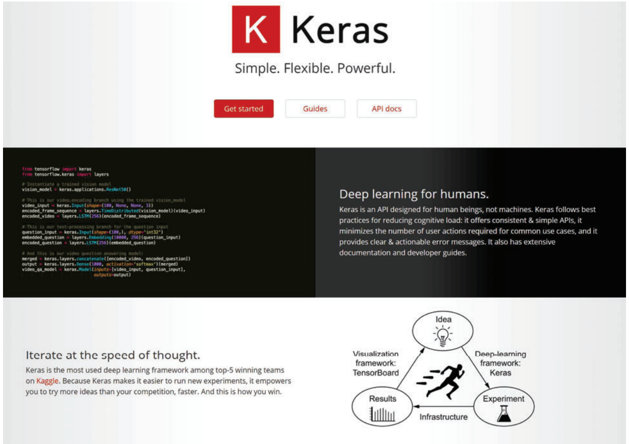
Confusion Matrix for Binary Classification

Evaluation is usually done with validation or test samples, but sometimes we evaluate for training samples

	$\hat{P}$ (predicted)	$\hat{N}$ (predicted)	
Positive P (actual) 20	True Positive TP 18	False Negative FN 2	Sensitivity/ Recall TP/(TP+FN)
Negative N (actual) 80	False Positive FP 5	True Negative TN 75	Specificity TN/(TN+FP)
	Precision TP/(TP+FP)	Accuracy (TP+TN)/(TP+TN+FP+FN)	

Refer to sklearn.metrics and Keras metrics

TensorFlow2 and Keras



Simple. Flexible. Powerful.

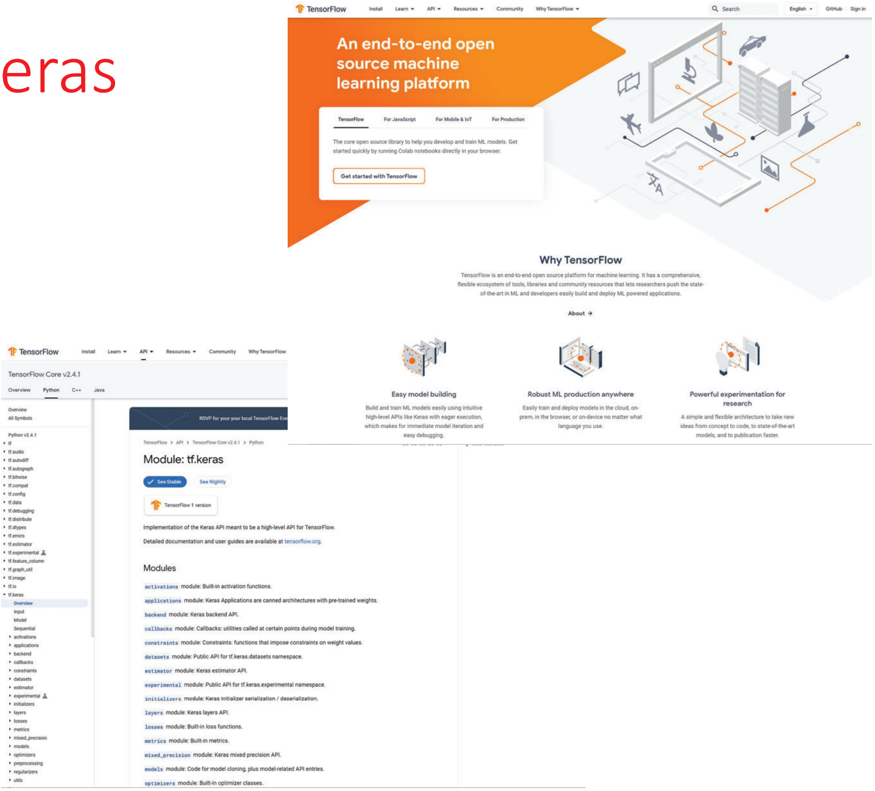
```
from tensorflow import keras
from tensorflow.keras import layers

# Instantiate a trained vision model
vision_model = keras.applications.ResNet50()

# This is our video.encoding branch using the trained vision_model
video_input = keras.Input(shape=(100, None, None, 3))
encoded_frame_sequence = layers.TimeDistributed(vision_model)(video_input)
encoded_video = layers.LSTM(256)(encoded_frame_sequence)

# This is our text-processing branch for the question input
question_input = keras.Input(shape=(100,), dtype='int32')
embedded_question = layers.Embedding(10000, 256)(question_input)
encoded_question = layers.LSTM(256)(embedded_question)

# And this is our video question answering model:
merged = keras.layers.concatenate([encoded_video, encoded_question])
output = keras.layers.Dense(1000, activation='softmax')(merged)
video_qa_model = keras.Model(inputs=[video_input, question_input],
                             outputs=output)
```

Deep learning for humans.

Keras is an API designed for human beings, not machines. Keras follows best practices for reducing cognitive load: it offers consistent & simple APIs, it minimizes the number of user actions required for common use cases, and it provides clear & actionable error messages. It also has extensive documentation and developer guides.

Iterate at the speed of thought.

Keras is the most used deep learning framework among top-5 winning teams on Kaggle. Because Keras makes it easier to run new experiments, it empowers you to try more ideas than your competition, faster. And this is how you win.



An end-to-end open source machine learning platform

The core open source library to help you develop and train ML models. Get started quickly by running Colab notebooks directly in your browser.

Why TensorFlow

TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications.

Easy model building

Build and train ML models easily using intuitive high-level APIs like Keras with eager execution, which makes for immediate model iteration and easy debugging.

Robust ML production anywhere

Easily train and deploy models in the cloud, on-prem, in the browser, or on-device no matter what language you use.

Powerful experimentation for research

A simple and flexible architecture to take new ideas from concept to code, to state-of-the-art models, and to publication faster.

TensorFlow Core v2.4.1

Overview
All Symbols

Python v2.4.1
- tf
- tf.audio
- tf.autodiff
- tf.autograph
- tf.bitwise
- tf.compat
- tf.config
- tf.data
- tf.debugging
- tf.distribute
- tf.dtypes
- tf.errors
- tf.estimator
- tf.experimental
- tf.feature_column
- tf.graph_util
- tf.image
- tf.io
- tf.keras
  - Overview
  - Input
  - Model
  - Sequential
  - activations
  - applications
  - backend
  - callbacks
  - constraints
  - datasets
  - estimator
  - experimental
  - initializers
  - layers
  - losses
  - metrics
  - mixed_precision
  - models
  - optimizers
  - preprocessing
  - regularizers
  - utils

TensorFlow > API > TensorFlow Core v2.4.1 > Python

Module: tf.keras

Implementation of the Keras API meant to be a high-level API for TensorFlow.

Detailed documentation and user guides are available at tensorflow.org.

Modules

`activations` module: Built-in activation functions.

`applications` module: Keras Applications are canned architectures with pre-trained weights.

`backend` module: Keras backend API.

`callbacks` module: Callbacks: utilities called at certain points during model training.

`constraints` module: Constraints: functions that impose constraints on weight values.

`datasets` module: Public API for tf.keras.datasets namespace.

`estimator` module: Keras estimator API.

`experimental` module: Public API for tf.keras.experimental namespace.

`initializers` module: Keras initializer serialization / deserialization.

`layers` module: Keras layers API.

`losses` module: Built-in loss functions.

`metrics` module: Built-in metrics.

`mixed_precision` module: Keras mixed precision API.

`models` module: Code for model cloning, plus model-related API entries.

`optimizers` module: Built-in optimizer classes.

Useful links:
https://keras.io
https://www.tensorflow.org
https://www.tensorflow.org/api_docs/python/tf/keras
https://www.tensorflow.org/guide/keras/sequential_model
https://www.tensorflow.org/tensorboard/get_started

Thank You

Questions?

Assoc Prof Tham Chen Khong

E-mail: eletck@nus.edu.sg