Fundamentals of Training AI Models for Computer Vision Applications

Amit Mate

Founder & CEO

GMAC Intelligence

2024 embedded VISION SUMMIT®

GMAC Intelligence

Content

- Vision AI Tasks
- Deep CNNs for Vision AI
- What is Training?
- Training vs Inferencing
- Types of Training
- Under the Hood – Model, Data, Process
- Training Frameworks and Tools
- Training a CNN in Keras
- Training Caveats
- Transfer Learning and Fine-tuning
- Data Augmentation
- Conclusions

Vision AI Tasks

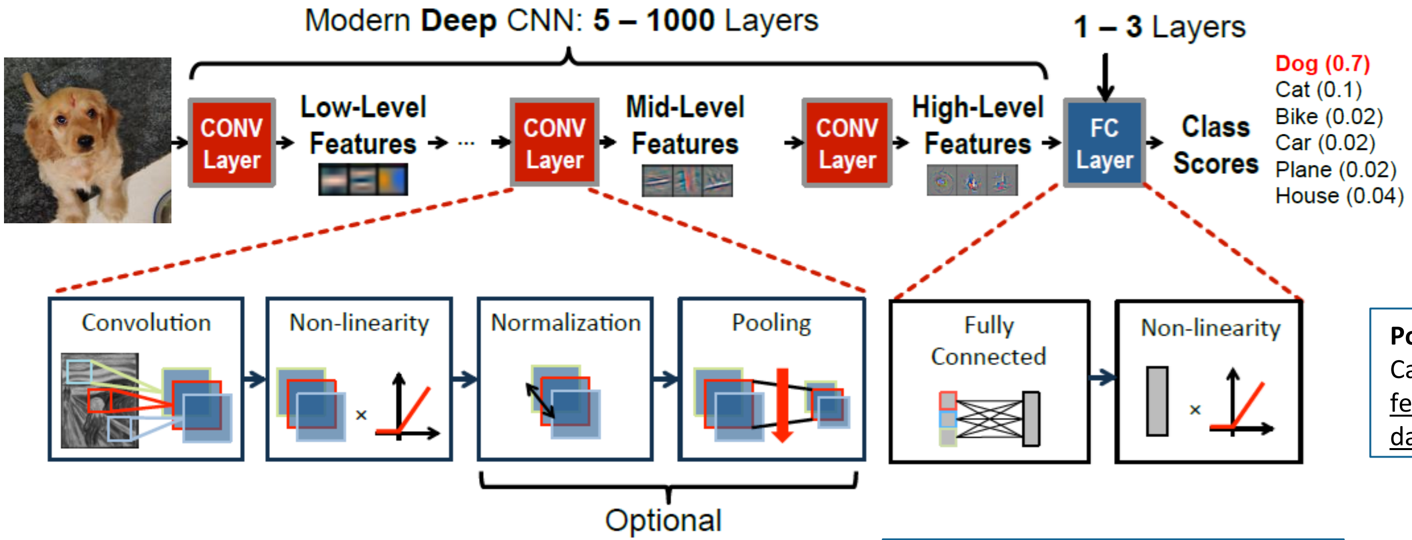
Classification

Object Detection

Segmentation

Caption Generation

Deep CNNs for Vision AI



Power of deep CNNs:
Capability of <u>learning features directly from visual data.</u>

Training CNNs:
CNNs learn these features during training process which is specific to the vision ai task.

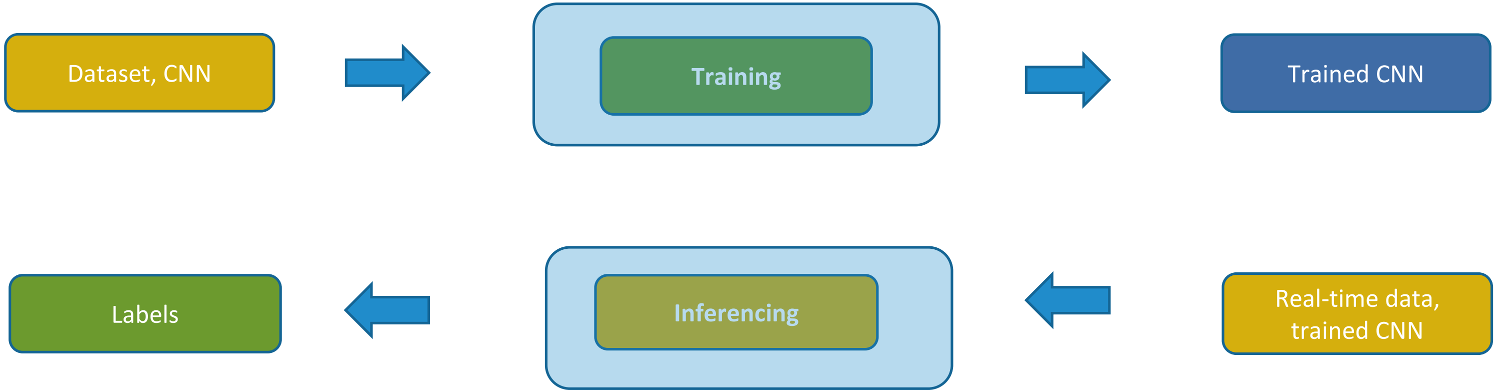
CNN parameters to be learned:
Convolution layer: kernels, bias
FC Layer: weights, bias
Normalization: mean, variance

Training 3Ws — What? Why? Where?

- What is training ?
  - It is the **process** of using **data** to **adjust** the **parameters** of the **model** such that it can make accurate predictions or inferences
- Why should we train ?
  - To make the model useful/accurate for executing (inferencing) a specific vision ai task
- Where should we train?
  - Usually* on a high-end server with GPUs or TPUs with high memory, storage and processing power

* Smaller models can be trained on PCs with GPUs

Training vs Inferencing



Training

- Offline, on high-end servers *
- Data limited
- Metrics: accuracy, generalization

Inferencing

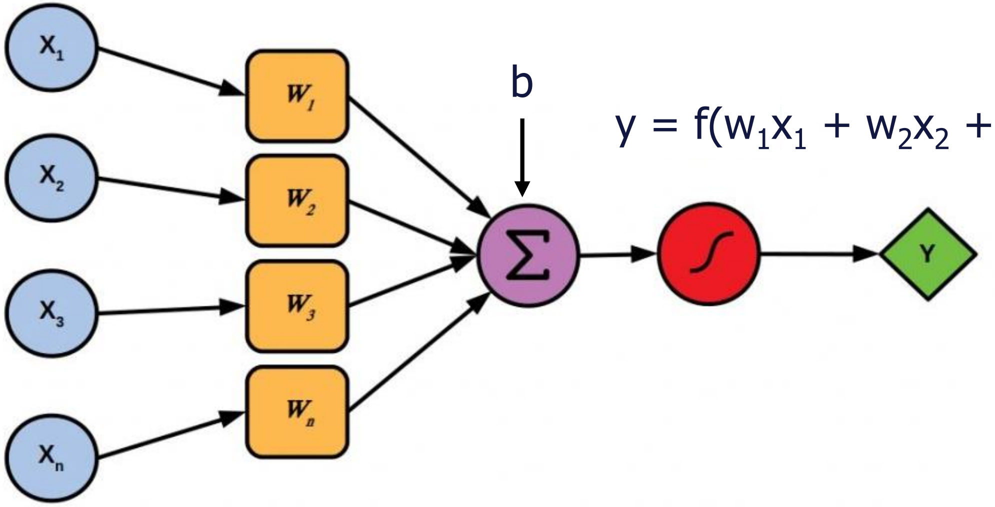
- Real-time, on **edge** devices *
- Memory, compute, storage limited
- Metrics: accuracy, latency

* Edge training and server inferencing also feasible

Training Methods

- **Supervised:** Model is trained on labeled data with input-output pairs
- **Unsupervised:** Model is trained on unlabeled data without any predetermined output
- **Semi-supervised:** Model is trained on both labeled and unlabeled data

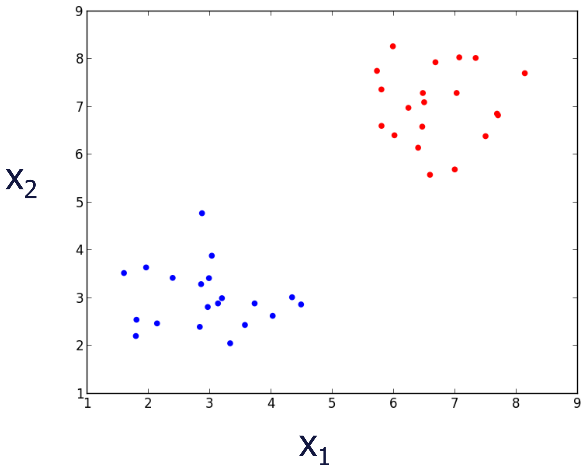
Perceptron Model

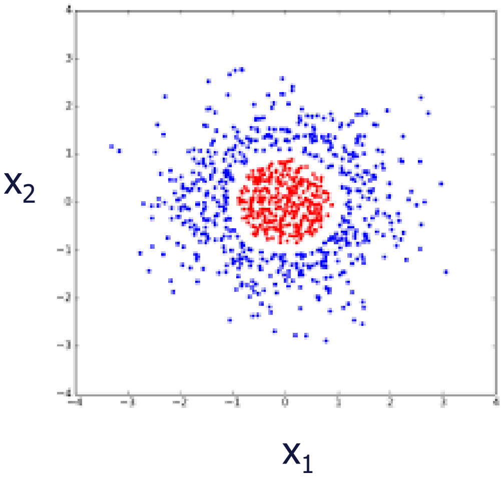


$$y = f(w_1x_1 + w_2x_2 + \ldots + w_nx_n + b)$$

Inputs Weights Sum Non-linearity Output

Data



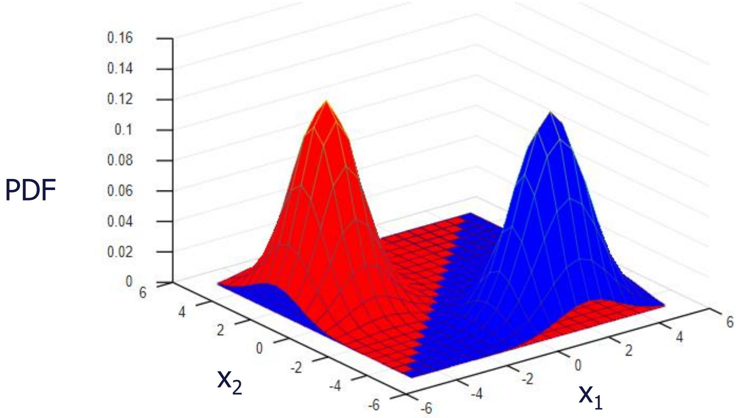


X: (x_1,x_2) => inputs

Y: (red, blue) => labels

Dataset: $(X,Y)_n$

Data



What is a good dataset ?

- Captures the underlying probability distribution of the data in real-world
- Accurate labels
- Well partitioned (training, validation, test)

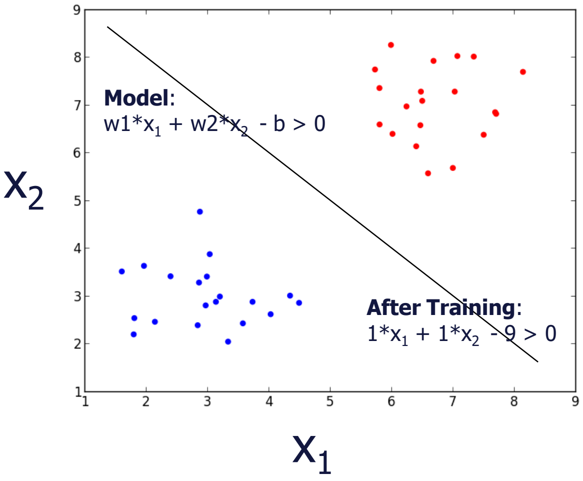
Dataset Partitions

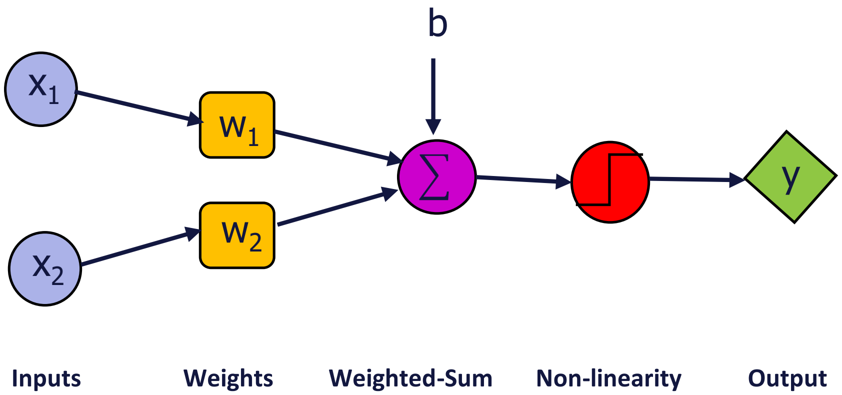
Training Set: Mutually exclusive subset of data used directly for learning parameters of the model during the training phase , typically 60-80% of the dataset, used for fitting model to the data.

Validation Set: Mutually exclusive subset of data used during learning phase for evaluation of the learned parameters , typically 10-20% of the dataset, used to prevent overfitting of the model.

Test Set: Mutually exclusive subset of data used after training is completed, typically 10-20% of the dataset, used for evaluation of the model on data which is not used for training the model

Learning





X: (x_1,x_2) => inputs
Y: (red = 0, blue =1) => labels
Dataset: $(X,Y)_n$

Learning Goal – Figure out b, w1 & w2 such that for any data point (x1,x2), model computes the label y accurately

Learning Algorithm

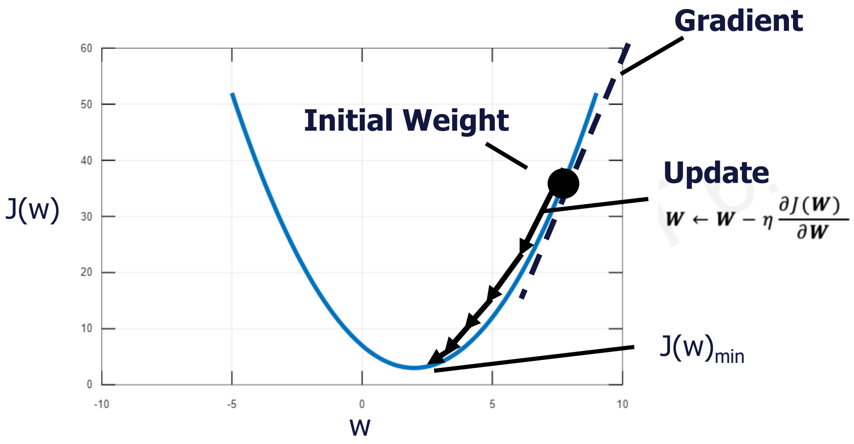
1. Assume random values for b, w1, w2
2. Iterate until **Y predicted correctly for "most" X in Dataset**
   - **Update** (b,w1,w2)

3. Use learned weights (b ,w1,w2) to classify X accurately

Learning via Optimization

Empirical Loss or Objective Function

$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}\big(\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}},\underbrace{y^{(i)}}_{\text{Actual}}\big)$$



Gradient Descent Algorithm

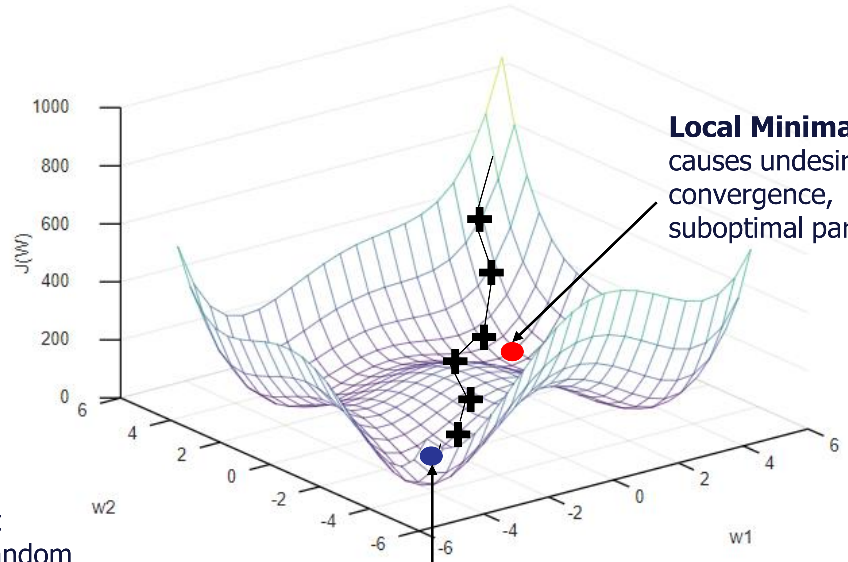
Stochastic Gradient Descent (SGD)

Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B}\sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights

Estimate of true gradient based on a batch "B" of random samples

Learning rate



Local Minima causes undesirable convergence, suboptimal parameters

Global Minima hard to converge on with a non-convex loss function

Key Parameters

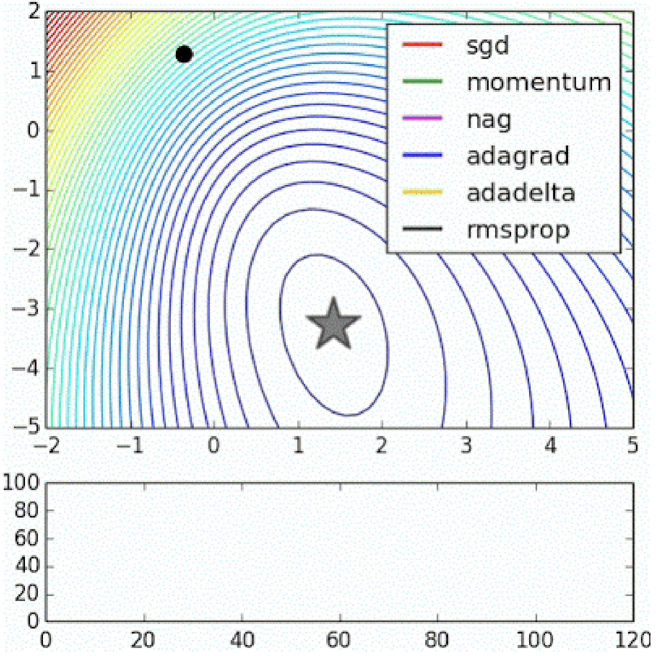
Training Dataset Size: The total number of data points used to train the model

Epoch: One full pass through the entire training dataset to update model weights

Batch Size: A subset of data points used for a single update of the model weights

Improvements on SGD

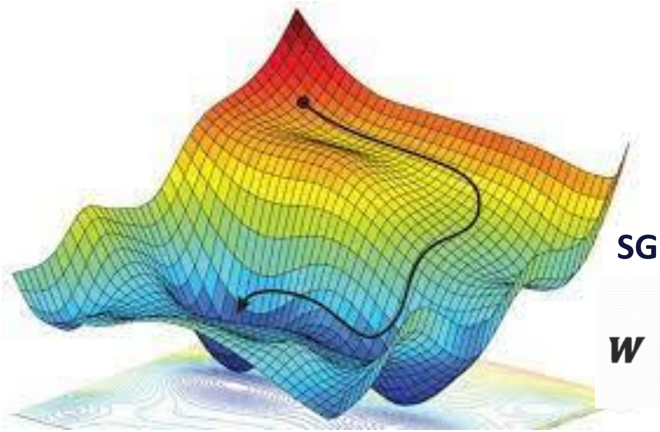
- **Adaptive Moment Estimation (Adam)**
  - Adaptive learning rate based on the momentum of gradients
  - Faster and more stable convergence
- **Root Mean Square Propagation (RMSprop)**
  - Adaptive learning rate based on moving average of the squared gradients
  - Mitigates the problem of exploding or vanishing gradients
- **Adagrad**
  - Adaptive learning rate based on historical gradient information
  - Reduces the learning rate for frequently occurring parameters



Animation from:
https://imgur.com/s25RsOr

Improvements on SGD

<u>Non-convex Loss Function Optimization</u>



SGD Update

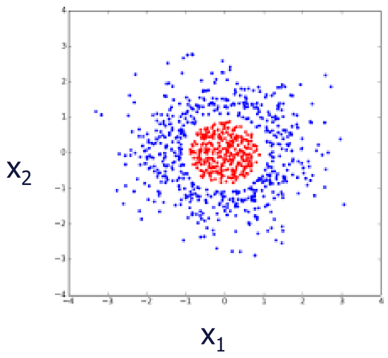
$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$$

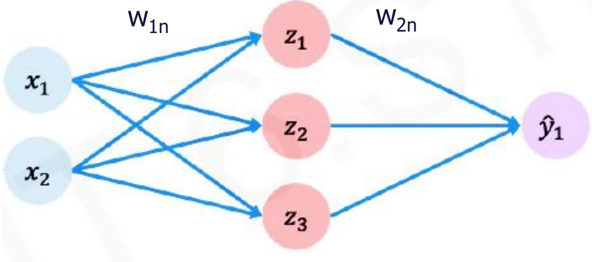
<u>Adam Update Rule Based on Moment "m"</u>

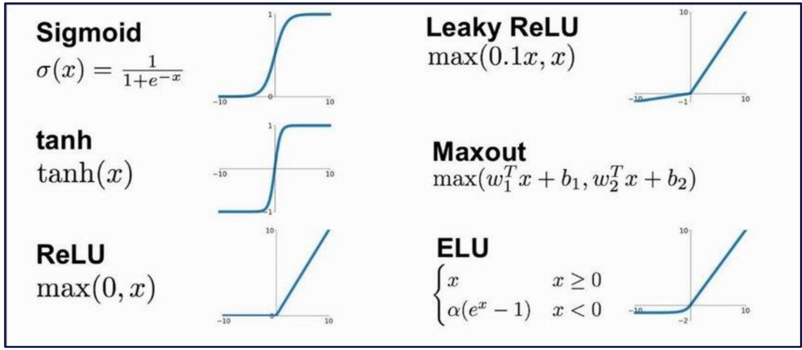
$v(t) = m*v(t\text{-}1) + (1 - m)*\partial J(\boldsymbol{W})/ \partial\, \boldsymbol{W}$

$\boldsymbol{W}(t) = \boldsymbol{W}(t\text{-}1) - \boldsymbol{\eta} * v(t)$

Nonlinearity Modelling







Nonlinearity

1. Non-linear relationships between input X and output Y needs multi-layer models and non-linear activation functions.

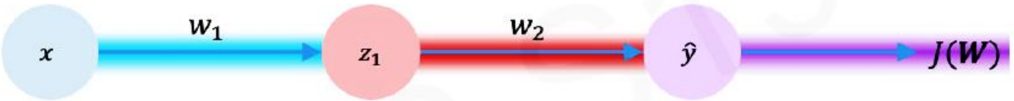
Multilayer Perceptron

2. Multi-layer model with multiple hidden layers for non-linear arbitrary function modelling. Multiple layers of weights need to be learned for accurate prediction.

Activation Functions

3. Choose functions based on problem type (binary or multi-class classification, regression). Needs experimentation.

Under the Hood — Backpropagation



$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * w_2 * x$$

Error backpropagation using chain rule of differentiation essential for learning parameters of a deep network

Training Resources for Beginners

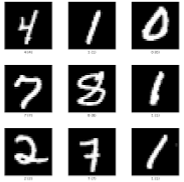
PyTorch

TensorFlow

K Keras

CIFAR-10

MNIST

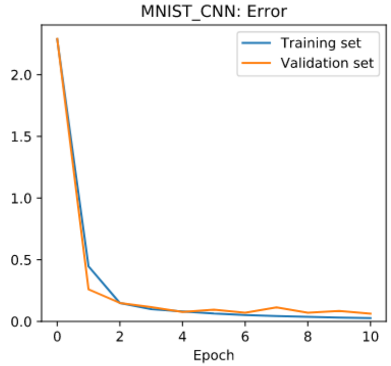


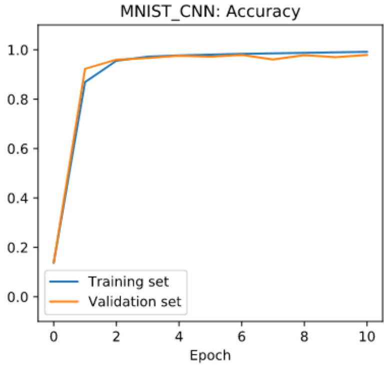
VOC-20

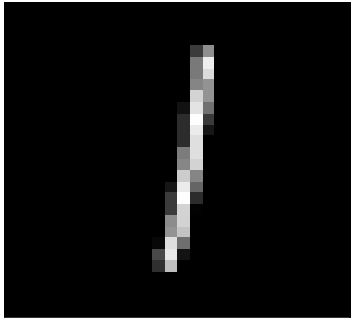
PASCAL2
Pattern Analysis, Statistical Modelling and Computational Learning

Training with Keras

```
# Load the data and split it between train and test sets
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Build the model
model = keras.Sequential(
    [
        keras.Input(shape=input_shape),
        layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
        layers.MaxPooling2D(pool_size=(2, 2)),
        layers.Flatten(),
        layers.Dropout(0.5),
        layers.Dense(num_classes, activation="softmax"),
    ]
)

# Train the model
model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])
model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1)

# Evaluate the trained model
score = model.evaluate(x_test, y_test, verbose=0)
print("Test loss:", score[0])
print("Test accuracy:", score[1])
```

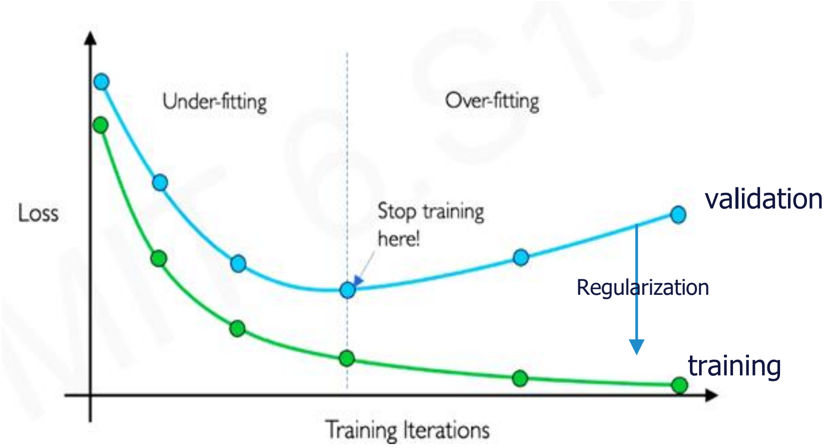


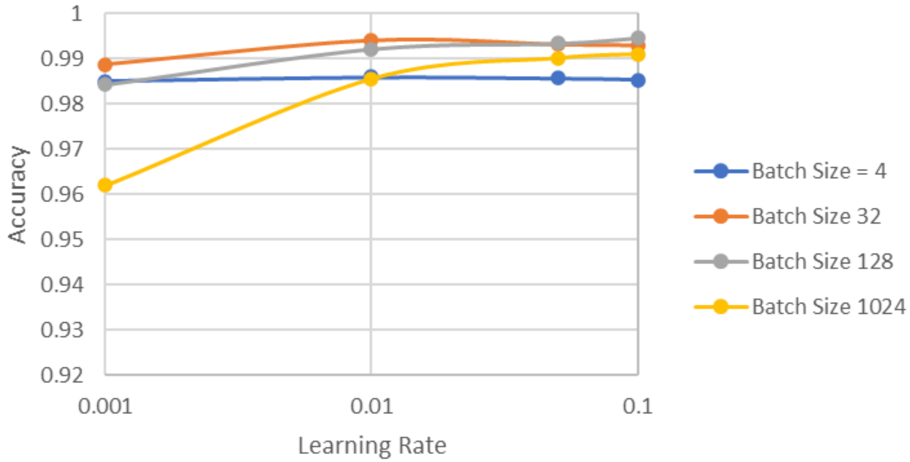




```
[5.2092713303864e-05,
 0.9586198329925537,
 0.006655485369265O795,
 0.00048394454643l303,
 0.01734444499015808,
 0.0013681561686098576,
 0.0008948856266215444,
 0.003324817866O8696,
 0.006120710633695126,
 0.005135755520313978]
```

Training Caveats





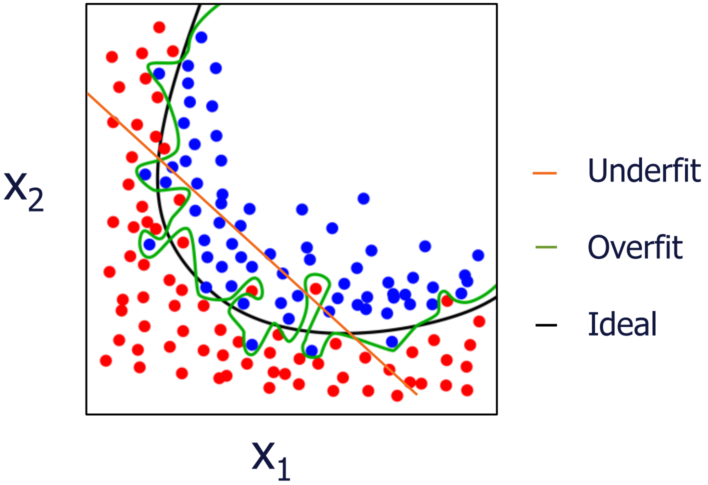
Caveats:

- Number of training epochs/iterations, dataset coverage, affects generalization and accuracy
- Learning rate, batch size, are important hyper-parameters for convergence and accuracy

Mitigation:

- Hyper-parameter tuning and/or heuristics
- Data augmentation and synthetic data
- Adjust network architecture (depth, width) to improve accuracy and convergence
- Regularization

Training Caveats — Regularization



Regularization Methods:

- Early termination
- L1/L2 (loss) regularization
- Dropout
- Batch normalization

L1/L2 Loss Regularization

<u>**Binary Cross Entropy Loss:**</u>

- **L1 Regularization (sparsity, less complexity)**

$$J(w) = -(1/N) \sum [y_i \log(\hat{y}_i) + (1- y_i) \log(1-\hat{y}_i)] + \lambda \, ||w||_1$$

- **L2 Regularization (smooth, less sensitive parameters, computationally efficient training)**

$$J(w) = -(1/N) \sum [y_i \log(\hat{y}_i) + (1- y_i) \log(1-\hat{y}_i)] + (\lambda/2) \, ||w||^2$$

Intuition: smaller values of "w" leads to better generalization, optimal λ for best fit (between overfitting and underfitting)

Dropout and Batch Normalization

Dropout

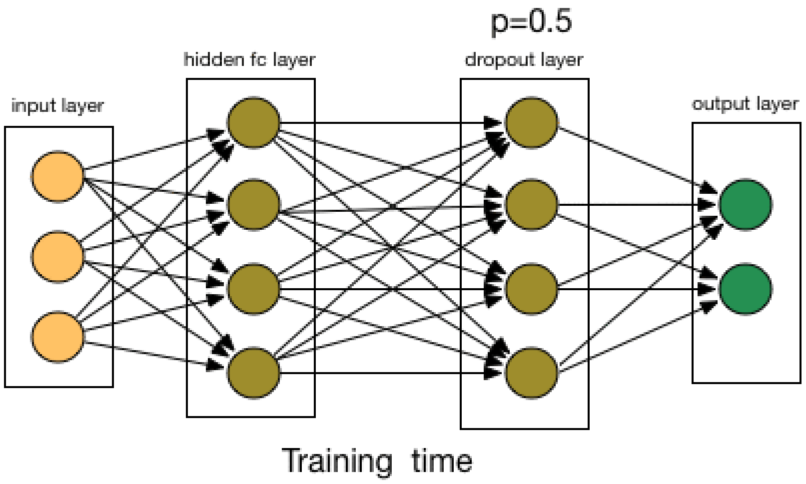


image source: primo.ai

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

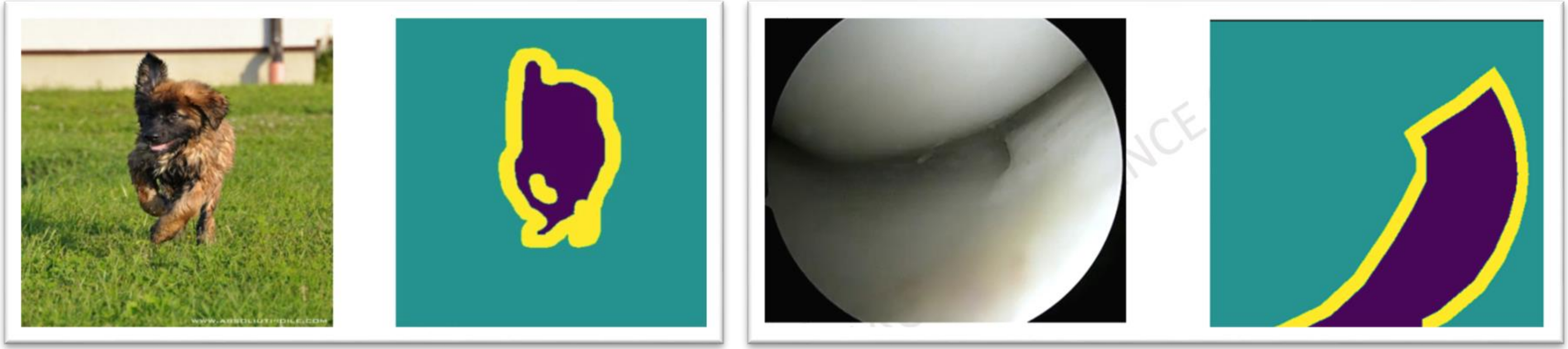
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Learned parameters: β, γ
Estimated parameters: μ , σ
Hyper parameter: ϵ

Transfer Learning

- **Transfer Learning:** it is the process of taking a model that has been trained on a large, comprehensive dataset for a particular task and then repurposing it for a second “unrelated” task (e.g., transfer learning applied from pet segmentation to orthoscopic tissue segmentation)



- Significantly reduce training time and computational resources needed
- Especially useful when target task has limited labelled data

Fine-tuning

- **Fine-tuning:** it is the process of taking a model that has been trained on a large, comprehensive dataset for a particular task and then tuning some layers to use it for a second "related" task.

- Generally used to improve accuracy of a deployed model to handle slightly different inputs not seen during training

Data Augmentation

- **Data Augmentation:** helps to improve the diversity/distribution of training dataset to match real-world scenarios. Techniques include rotations, translations, flipping, scaling, and changes in brightness or contrast for images. Improves generalization of the model, prevents overfitting and makes models more robust.

Conclusions

- Trained deep CNNs can accomplish various vision AI tasks
- Key ingredients for training CNNs: dataset, learning algorithm, back-propagation
- A good dataset should be well-partitioned and represent the underlying distribution of data
- A good training algorithm is efficient in learning parameters from data
- Accuracy and generalization are KPIs of a well-trained network
- Leverage transfer-learning, heuristics and regularization to make training more efficient
- Keras, Tensorflow and Pytorch are good frameworks to start training

Further Resources

- Keras https://keras.io/
- Tensorflow https://www.tensorflow.org/
- Pytorch https://pytorch.org/
- Colab Online Training Servers https://colab.research.google.com/
- SOTA Vision Models https://paperswithcode.com/area/computer-vision
- MIT Deep Learning Course http://introtodeeplearning.com/