# Introduction

Arthur Samuel(1959. Machine learning: Field of study that gives computers the ability to learn without being explicitly programmed.

Tom Mitchell(1998) Well-posed learning problem: A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T as measured by P improves with experience E.

## Machine learning algorithms:

* Supervised learning
* Unsupervised learning

Others: Reinforcement learning, recommender system.

## Types of ML Algorithms:

-Supervised: train on labelled data to predict labels.

- Self-supervised: labels are generated from data(E.g. Word2vec, BERT).

-Semi-supervised: train on both labels and unlabeled data, learn models to predict labels or infer labels for unlabeled data.

- E.g. self-training

-Unsupervised: train on unlabeled data.

- E.g. clustering, density estimation(GAN).

-Reinforcement: Use observations from the interaction environment to take actions to maximise reward.

# Supervised learning

## Components in Supervised Training:

-Model: output predicted from inputs.

- E.g. listing house— sale price.

-Loss: Measure difference between predicted and ground truth labels.

- E.g. (predict\_price - sale\_price)^2

- Objective: any function to optimise during training.

- E.g. minimise the sum of losses over examples.

-Optimization: learn model parameters by solving the objective function.

## Types of Supervised Model:

-Decision tree: use trees to make decisions.

-Linear methods: decision is made from a linear combination of input.

-Kernel machines: use kernel functions to compute feature similarities.

-Neural networks: use neural networks to learn feature representations.

## Decision tree:

Pors:

* Explainable.
* Can handle both numerical and categorical features.

Cons:

* Very non-robust (ensemble to help).
* Complex trees cause overfitting (prune trees)
* Not easy to be parallelized in computing.

### Random forest:

* Train multiple decision trees to improve robustness.

1. Each tree is trained independently.
2. Majority voting for classification, average for regression.

* Where is the random forest from

1. Bagging: randomly sample training examples with replacement.
2. E.g. [1,2,3,4,5]~~[1,2,2,3,4].

b. randomly select a subset of features.

### Gradient Boosting Decision Trees:

* Train multiple trees sequentially.
* At step t = 1, …, denote by  the sum of past trained trees.

1. Train a new tree  on residuals: ,...
2. 

* The residual equals to –  if using the mean square error as the loss, so it’s called gradient boosting.

### Summary：

* Decision tree: an explainable model for classification/regression.
* Ensemble tree to reduce bias and variance.

1. Random forest: trees trained in parallel with randomness.
2. Gradient boosting tree: train in sequential on residuals.

* Trees are widely used in industry.

1. Simple, easy-to-tune, often gives satisfied results.

## Linear regression:

* A simple house price prediction:

1. Assume 3 features: .
2. The predicted price is .
3. Weights  and bias b will be learnt from training data.

* In general, given data , linear regression predicts



1. Trainable weights w = [w1,w2, …wp] and bias b.

# weight w has shape (p, 1)

# bias b is a scalar

# data X has shape (p, 1)

y\_hat = (x\*w).sum() + b

### Objective function:

* Collect n training examples , with labels .
* Objective: minimise the mean square error (MSE),



* Exercise: write the closed form solution.

### Use linear regression for classification:

* Regression: continuous output in R.
* Multi-class classification:

1. Vector outputs, i-th output reflects the confidence score as belongs to class i.
2. Learn a linear model for each class 
3. Label  where  if i = y otherwise
4. Minimise MSE loss 
5. Predict label 

### Softmax Regression：

* Label where  if  otherwise 0.
* Turn scores into probabilities (non-negative, sum to 1)

 where 

0\_exp = torch.exp(0)

partition = 0\_exp.sum(

1, keepdim=True)

Y = 0\_exp / partition

1. Still a linear model, decision made as 

* Cross-entropy loss between two distributions  and 



1. When label class is , assigns less penalty on  as long as .

### Mini-batch stochastic gradient descent (SGD):

* Train by mini-batch SGD (by various other ways as well)

1. **w** model param, **b** batch size,  learning rate at time **t.**
2. randomly initialise 
3. repeat **t = 1, 2, …** until converge
4. randomly samples  with 
5. update .

* Pros: solve all objective in this course except for trees
* Cons: sensitive to hyper-parameters  and .

#### Code:

* Train a linear:

1. Regression model with mini-batch SGD

* Hyperparameter

1. batch\_size
2. Learning\_rate
3. num\_epochs
4. Full code at: http://d2l.ai/chapter\_linear-networks/linear-regression-scratch.html

# ‘features’ shape is (n, p), ‘labels’ shape is (p, 1)

def data\_iter(batch\_size, features, lables):

num\_examples = len(features)

indices = list(range(num\_examples))

for i in range(0, num\_examples, batch\_size):

batch\_indices = torch.tensor(

indices[i:min(i + batch\_size, num\_examples)])

Yield features[batch\_indices], labels[batch\_indices]

w = torch.normal(0, 0.01, size=(p, 1), requires\_gard=True)

b = torch.zeros(1, requires\_grad=True)

for epoch in range(num\_epochs):

for X, y in data\_iter(batch\_size, features, labels):

y\_hat = X @ w + b

loss = ((y\_hat - y)\*\*2 / 2).mean()

loss.backward()

for param in [w, b]:

partam -= learning\_rate \* param

param.grad.zero\_()

### Summary:

* Linear methods linearly combine inputs to obtain predictions
* Linear regression used MSE as the loss function.
* Softmax regression is for multiclass classification

1. Turn predictions into probabilities and use cross-entropy as loss

* Mini-batch SGD can learn both models (and later neural networks as well)

# Neural networks

Handcrafted features  learned features.

Raw dataneural networks linear/softmax regressionoutput

* NN usually requires more data and more computation.
* NN architectures to model data structures

1. Multilayer perce
2. Convolutional neural networks
3. Recurrent neural networks
4. Transformer.

## Linear methods Multilayer perceptron (MLP)

* A dense (fully connected, or linear) layer has parameters , it computes output ,

1. Linear regression: dense layer with 1 output.
2. Softmax regression: dense layer with **m** outputs + softmax.

### MLP

* Activation is an elemental-wise non-linear function.

1. , 
2. It leads to non-linear models.

* Stack multiple hidden layers (dense + activation) to get deeper models.
* hyper-parameters

1. # hidden layers,
2. # outputs for each hidden layer.

#### Code

* MLP with 1 hidden layer.
* Hyperparameter: num\_hiddens

def relu(X):

Return torch.max(X,0)

W1 = nn.Parameter(torch.randn(num\_inputs, num\_hiddens) \* 0.01)

b1 = nn.Parameter(torch.zeros(num\_hiddens))

W2 = nn.Parameter(torch.randn(num\_hiddens, num\_outputs) \* 0.01)

b2 = nn.Parameter(torch.zeros(num\_outputs))

H = relu(X @ W1 + b1)

Y = H @ W2 +b2

### Dense layer Convolution layer

* Learn ImageNet (300\*300 images with 1K classes) by a MLP with a single hidden layer with 10K outputs.

1. It leads to 1 billion learnable parameters, that’s too big!
2. Fully connected: an output is a weighted sum over all inputs.

* Recognize objectives in images

1. Translation invariance: similar output no matter where the object is
2. Locality: pixels are more related to near neighbours.

* Build the prior knowledge into the model structure.

1. Achieve same model capacity with less # params.

### Convolution layer

* Locality: an output is computed from  input windows.
* Translation invariant：outputs use the same  weights (kernel).
* # model params of a conv layer does not depend on input/output sizes.
* A kernel learns to identify a pattern.

#### Code

* Convolution with single input and output channels

# both input ‘X’ and weight ‘K’ are matrices

h, w = K.shape

Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))

for i in range(Y.shape[0]):

For j in range(Y.shape[1]):

Y[i, j] = (X[i:i + h, j:j + w] \* K).sum() # precisely it’s called cross-correlation.

### Pooling layer

* Convolution is sensitive to location

1. A pixel shift in the input results in a pixel shift in output

* A pooling layer computes mean/max in  windows.

# h, w: pooling window height and weight

# mode: max or avg

Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))

for i in range(Y.shape[0]):

for j in range(Y.shape[1]):

If mode == ’max’:

Y[i, j] = X[i : i+h, j : j+w].max()

elif mode == ‘avg’:

Y[i, j] = X[i : i+h, j : j+w].mean()

The input picture will be shifted. To solve this problem, that is to increase the robustness of the model's impact on the displacement, the concept of pooling layer is introduced. Every time the data is read, h rows and w columns are read more, so that the model can resist a certain amount. The displacement changes within the range.

### Convolutional Neural Networks (CNN)

* A neural network uses a stack of convolution layers to extract features.

1. Activation is applied after each convolution layer.
2. Using pooling to reduce location sensitivity.

* Modern CNNs are deep neural network with various hyper-parameters and layer connections (AlexNet, VGG, Inceptions, ResNet, MobileNet)

### Dense layer Recurrent networks

* Language model: predict the next word.
* Use MLP naively doesn’t handle sequence info well

#### RNN and Gated RNN

* Simple RNN: 
* Gated RNN (LSTM, GRU): finer control of information flow

1. Forget input: suppress  when computing 
2. Forget input: suppress  when computing 

* Code

1. Implement simple RNN

W\_xh = nn.Parameter(torch.randn(num\_inputs, num\_hiddens) \* 0.01)

W\_hh = nn.Parameter(torch.rand(num\_hiddens, num\_hiddens) \* 0.01)

b\_h = nn.Parameter(torch.zero(num\_hiddens))

H = torch.zeros(num\_hiddens)

outputs = []

for X in inputs: # ‘inputs’ shape : (num\_steps, batch-size, num\_inputs

H = torch.tanh(X @ W\_xh + H @ W\_hh + b\_h)

outputs.append(H)

Full code at: http://d2l.ai/chapter\_recurrent-neural-networks/rnn-scratch.html

## Summary

* MLP: stack dense layers with non-linear activations
* CNN: stack convolution activation and pooling layers to efficiently extract spatial information.
* RNN: stack recurrent layers to pass temporal information through hidden state.

# 

# Model Validation

### Model Metrics

* In supervised learning, we train models by minimizing the training loss.

1. The loss value is a widely used metric to measure model quality.

* There are many other metrics

1. Model specific: e.g. accuracy for classification, mAP for objective tection
2. Business specific: e.g. revenue, inference latency

* We select models by multiple metrics.

1. Just like how you choose cars.

### AUC & ROC

* AUC, the area under the ROC curve, measures the chance a model can distinguish classes.

1. Choose various theta, predict as pos hat y >= theta else neg.

## Underfitting & Overfitting

* Training error: model error on the training data.
* Generalisation error: model error on new data.
* Example: practice a future exam with past exams.

1. Doing well on past exams (training error) doesn’t guarantee a good score on the future exam (generalisation error).
2. Student A gets 0 error on past exams by rote learning.
3. Student B understands the reasons for given answers.

|  | Training error |
| --- | --- |
| Generalisation error | |  | Low | High | | --- | --- | --- | | Low | Good | Bug? | | High | Overfitting | Underfitting | |

| Data and Model Complexity | Data complexity |
| --- | --- |
| Model complexity | |  | Low | High | | --- | --- | --- | | Low | Normal | Underfitting | | High | Overfitting | Normal | |

### Model Complexity

* The ability to fit variety of functions

1. Low complexity models struggles to fit training sets.
2. High complexity models can memorize the training set

* It’s hard to compare between very different algorithms.

1. E.g. trees vs neural network.

* In an algorithm family, two factors matter:

1. The number of parameters.
2. The values taken by each parameter.

### Data Complexity

* Multiple factors matter.

1. # of examples.
2. # of elements in each example.
3. time/space structure.
4. diversity.

* Again, hadr to compare among very different data

1. E.g. a char vs a pixel.

### Model Selection

* Pick a model with a proper complexity for your data.

1. Minimize the generalisation error
2. Also consider business metrics.

* Start with pick up a model family, then select proper hyperparameters

1. Trees: # trees, maximal depths.
2. Neural networks: architecture, depth(#layers), width(#hidden hits), regularizations.

### Summary

* We care about generalisation errors.
* Model complexity: the ability to fit various functions
* Data complexity: the richness of information.
* Model selection: match model and data complexities.

## Model Validation

* Approximated by the error on a test dataset, which can be only used once.

1. Your score on the midterm exam.
2. The house sale price 1 bided.
3. Dataset used inprivate leaderboard in Kaggle.

* Validation dataset: can be used multiple times.

1. Often a part of the training dataset.
2. When we use “test”, in more time we mean “validation”

### Hold Out Validation

* Split your data into “train” and “valid” sets (often calls “test”)

1. Train your model on the train set, use the error on the valid set to approximate the generalisation error.

* Often randomly select n% examples as the valid set.

1. Typical choices n = 50, 40, 30, 20, 10

### Split non I.I.D. data

* Random splitting may not work.
* Sequential data: e.g. house sales, stock prices.

1. Valid set should be after the train set.

* Examples belong to grounds: e.g. photos of the same person.

1. Split among groups instead of examples.

* Sample more from minor classes.

### K-fold Cross Validation

* Useful when not sufficient data.
* algorithm:

1. Partition the training data into K parts.
2. For i = 1, …, K
3. Use the i-th part as the validation set, the rest for training.
4. Report the average K validation errors.

* Popular choices: K = 5 or 10

### Common Mistakes

* Personal opinion: 90% super good results in ML are due to bugs.

1. Contaminated valid set is the #1 reason.

* Valid set has examples from train set.

1. Duplicated examples in original data.
2. Often happens when integrating data.
3. Scrape images from search engines to evaluate models trained on ImageNet.

* Information leaking.

1. Often happen for non l.l.D data: e.g. use future to predict person’s face before

### Summary

* The test data is used once to evaluate your model
* Sample a validation set from the training data to approximate the test data.

1. You can evaluate multiple times to select models
2. Validation data should be close to the test data, especially for I.I.D data.
3. Improper valid sets are a common mistake that lead to false good results.

## Bias & Variance

* In statistic learning, we measure a model in terms of bias and variance.

### Bias-Variance Decomposition

* Sample data  from .
* Learn  from D by minimizing MSE, we want it to generate well over different choices of D.



= 

= 

, varepsilon is independent of f hat.

### Reduce Bias & Variance

* Reduce bias

1. A more complex model
2. Boosting
3. Stacking

* Reduce variance

1. A simple model
2. Regularization
3. Bagging
4. Stacking

* Reduce 

1. Improve data

(boosting, bagging, stacking)Ensemble learning: use multiple models to improve predictive performance.

### Summary

* Decompose model generalisation error into bias, variance and intrinsic error.
* Ensemble learning combines multiple models to reduce both bias and variance.

## Bagging

Bagging - Bootstrap AGGrgratING

* Bagging trains n base learners in parallel.
* Make decisions by averaging learners’ outputs (regression) or majority voting (classification)
* Each learner is trained on data by bootstrap sampling.

1. Assume m training examples, then randomly sampling m examples with replacement.
2. Around 1- 1/e ==63% examples will be sampled, the rest (out of bag) can be used for validation.

### Bagging code

class Bagging:

def \_\_init\_\_(self, base\_learner, n\_learners):

self.learners = [clone(base\_learner) for \_ in range(n\_learners)]

def fit(self, X, y):

for learner in self.learners:

examples = np.random.choice(

np.arange(len(X)), int(len(X)), replace=True)

learner.fit(X.iloc[examples, :], y.iloc[examples])

def predict(self, X):

preds = [learner.predict(X) for learner in self.learner)

Return np.array(preds).mean(axis=0)

### Random forest

* Useb decision tree as the base learner.
* Often randomly select a subset of features for each learner.
* Results on house sale data.

### Unstable learner

* Bagging reduces variance, especially for unstable learners.
* Consider regression for simplicity, given ground truth f and base learner h, bagging: .
* Given , we have 

1. Bagging reduces more variance when base learners are unstable.

### Summary

* Bagging train multiple learner on data by bootstrap sampling.
* Bagging reduce variance, especially for unstable learners.

## Boosting

* Boosting combines weak learners into a strong one

1. Primarily to reduce bias.

* Learn n weak learners sequentially, at step i:

1. Train a weak learner , evaluate its errors .
2. Re-sample data according to  to focus on wrongly predict samples

* Notable examples include AdaBoost, gradient boosting.

### Gradient Boosting

* Denote by  the model at time t with 
* At step t = 1

1. Train a new model  on residuals: 
2. 
3. The learning rate regularizes the model by shrinkage.

* The residuals equal to  if using MSE as the loss

1. Other boosting algorithms (e.g. AdaBoost) can also be gradient descent in the function space.

### Gradient Boosting Code

Class GradientBoosting:

def \_\_init\_\_(self, base\_learner, n\_learners, learning\_rate):

self.learners = [clone(base\_learner) for - in range(n\_learners)]

self.lr = learning\_rate

def fit(self, X, y):

residual = y.copy()

for learner in self.learners:

learner.fit(X, residual)

residual -= self.lr \* learner.predict(X)

def predict(self, X):

preds = [learner.predict(X) for learner in self.learners

return np.array(preds).sum(axis=0) \* self.lr

### Gradient Boosting Decision Tree(GBDT)

* Use decision tree as the weak learner.

1. Regularise by a small max\_depth and randomly sampling features.

* Sequentially constructing trees runs slow.

1. Popular libraries use accelerated algorithms, e.g. XCBoost, lightGBM

### Summary

* Boosting combines weak learners into a strong one to reduce bias.
* Gradient boosting learners weak learners by fitting the residuals.

## Stacking

* Combine multiple base learners to reduce variance.

1. Base learners can be different model types.
2. Linearly combine base learners outputs by learned parameters.

* Widely used in competitions.
* In comparison, bagging.

1. Uses same type models.
2. Uses boostrap to get diversity.

### Multi-layer Stacking

* Stacking base learners in multiple levels to reduce bias.

1. Can use a different set of base learners at each level.

* Upper levels (e.g. L2) are trained on the outputs of the below levels (e.g. L1).

1. Concatenating original inputs helps.

### Overfitting in Multi-layer Stacking

* Train learner from different levels on different data to alleviate overfitting.

1. Split training data into A and B, train L1 learners in A, predict on B to generate inputs to L2 learners.

* Repeated k-fold bagging:

1. Train k models as in k- fold cross validation.
2. Combine predictions of each model on out-of-fold data
3. Repeat step 1,2 by n time, average the n predictions of each example for the next level training.

### Summary

* Stacking combine multiple learners to reduce variance
* Stacking learners in multiple levels to reduce bias.

1. Repeated k- fold bagging: fully utilise data and alleviate overfitting:

## Model Combination Summary

* The goal is to reduce bias and variance.

| Reduce | Bias | Variance | Computation Cost | Parallelization |
| --- | --- | --- | --- | --- |
| Bagging |  | Y | n | n |
| Boosting | Y |  | n | l |
| Stacking |  | Y | n | n |
| K-fold multi-level stacking | Y | Y | n\*l\*k | n\*k |

N: number of learners, L: number of levels, K: k-fold.

# Model Tuning

## Model tuning

### Manual Hyperparameter Tuning

* Start with a good baseline, e.g. default settings in high-quality toolkits, values reported in papers.
* Tune a value, retrain the model to see the changes.
* Repeat multiple times to gain insights about.

1. Which hyperparameters are important.
2. How sensitive the model to hyperparameters.
3. What are the good ranges.

* Needs careful experiment management.
* Save your training logs and hyperparameters to compare, share and reproduce later.

1. The simplest way is saving logs in next and putting key metrics in Excel.
2. Better options exist, e.g. tensorboard and weights & bias

* Reproducing is hard, it relates to:

1. Environment ( hardware & library).
2. Code.
3. Randomness (seed).

### Automated Hyperparameter Tuning

* Computation costs decrease exponentially, while human costs increase.
* Cost per training for a typical ML task:

1. E.g. 1M user logs, 10K images.

* Cost of a data scientist per day > $500
* Usd algorithms if it outperforms human after 1000 trials.

1. Typically beat 90% data scientists.

### Automated Machine Learning (AutoML)

* Automate every step in applying ML to solve real-world problems: data cleaning, feature extraction, model selection…
* Hyperparameter optimization (HPO):
* Find a good set of hyperparameters through search algorithms.
* Neural architecture search (NAS):

1. Construct a good neural network model.

## HPO algorithms

### Search Space

* Specify range for each hyperparameter

| Hyper-Parameter | Range | Distribution |
| --- | --- | --- |
| Model (backbone) | [MobileNetV2\_0.25，MobileNetV3\_small，MobileNetV3\_large，ResNet18\_V1b，ResNet18\_V1b，ResNet34\_V1b，ResNet50\_V1b,ResNet101\_V1b,VGG16\_bn，se\_ResNext50\_32\*4d，ResNest50，ResNest200 ] | categorical |
| Learning rate\* | [1e-6，1e-1] | log-uniform |
| Batch size\* | [8，16，32，64，128，256，512 ] | categorical |
| momentum\*\* | [0.85，0.98] | uniform |
| Weight decay\*\* | [1e-6， 1e-2] | log-uniform |
| detector | [Faster-RCNN, SSD, YOLO-V3, Center-Net] | categorical |

* The search space can be exponentially large.

1. Need to carefully design the space to improve efficiency.

### HPO algorithms: Black-box or Multi-fidelity

* Black-box: treats a training job as a black-box in HPO:

1. Completes the training process for each trial.

* Multi-fidelity: modifies the training job to speed up the search.

1. Train on subsampled datasets.
2. Reduce model size (e.g. less #layers, #channels)
3. Stop bad configuration earlier.

### Two most common HPO strategies

* Grid search

1. Code:

for config in search\_space:

train\_and\_eval(config)

return best\_result

1. All combinations are evaluated.
2. Guarantees the best results.
3. Curse of dimensionality.

* Random search

1. Code:

for \_ in range(n):

config = random\_select(search\_space)

train\_and\_eval(config)

return best\_result

1. Random combinations are tried.
2. More efficient than grid search.

### Bayesian Optimization (BO)

* BO: Iteratively learn a mapping from HP to objective function.

Based on previous trials. Select the next trial based on the current estimation.

* Surrogate model

1. Estimate how the objective function depends on HP.
2. Probabilistic regression models: Random forest, Gaussian process.

* Acquisition max means uncertainty and predicted objectives are high.

1. Acquisition max means uncertainty and predicted objectives are high.
2. Sample the trial according to the acquisition function.
3. Trade off exploration and exploitation.

* Limitation of BO:

1. In the stages, similar to random search.
2. Optimization process is sequential.

### Successive Halving

* Save the budget for the most promising config.
* Randomly pick n configurations to train m epochs.
* Repeat until one configuration left:

1. Keep the **best** n/2 configuration to train another m epochs.
2. Keep the **best** n/4 configuration to train another 2m epochs.

* Select n and m based on training budget and #epoch needed for a full training.

### Hyperband

* In Successive Halving

1. n: exploration
2. m: exploitation

* Hyperband runs multiple Successive Halving, each time decreases n and increases m.

1. More exploration first, then do more exploit.

### Summary

* Black-box HPO: grid / random search, bayesian optimization.
* Multi-fidelity HPO: successive Halving, Hyperband.
* In practice, start with random search.
* Beware there are top performers.

1. You can find them by mining your training logs, or what common configurations used in paper / code.

## NAS algorithms

### Neural Architecture Search (NAS)

* A neural network has different types of hyperparameters:

1. Topological structure: resnet-ish, mobilenet-ish, #layers
2. Individual layers: kernel\_size, # channels in convolutional layer, #hidde\_outputs in dense/recurrent layers.

* NAS automates the design of neural network.

1. How to specify the search space of NN.
2. How to explore the search space.
3. Performance estimation.

### NAS with Reinforcement Learning

* Zoph & Le 2017

1. A RL-based controller (REINFORCE) for proposing architecture.
2. RNN controller outputs a sequence of tokens to config the model architecture.
3. Reward is the accuracy of a sample model at convergence.

* Naive approach is expensive and sample inefficient (~2000 GPU days). To speed up NAS:

1. Estimate performance.
2. Parameter sharing (e.g. EAS, ENAS)

### The One-shot Approach

* Combines the learning of architecture and model params.
* Construct and training a single model presents a wide variety of architectures.
* Evaluate candidate architectures.

1. Only care about the candidate ranking.
2. Use a proxy metric: the accuracy after a few epochs

* Re-train the most promising candidate from scratch.

#### Differentiable Architecture Search

* Relax the categorical choice to a softmax over possible operation:

1. Multiple candidates for each layer.
2. Output of i-th candidate at layer l is .
3. Learn mixing weights . The input for i + 1 -the layer is  with .
4. Choose candidate .
5. Jointly learn  and network paRAMETERS.

* A more sophisticated version (DARTS) achieves SOTA and reduces the search time to ~3 GPU days.

### Scaling CNNs

* A CNN can be scaled by 3 ways:

1. Deeper: more layers.
2. Wider: more output channels.
3. Larger inputs: increase input image resolutions.

* EfficientNet proposes a compound scaling.

1. Scale depth by , width by , resolution by .
2. , so increase FLOP by 2x if =1.
3. Tune .

### Research directions

* Explainability of NAS result.
* Search architecture to fit into edge devices.

1. Edge devices are more and more powerful, data privacy concerns.
2. But they are very diverse (CPU/GPU/DSP, 100x performance difference) and have power constraints.
3. Minimise both model loss and hardware latency.
4. E.g. minimise loss X log(latency)B

* To what extent can we automates the entire ML

### Summary

* NAS searches a NN architecture for a customizable goal

1. Maximise accuracy or meet latency constraints on particular hardware.

* NAS is practical to use now:

1. Compound depth, width, resolution scaling.
2. Differentiable one-hot neural network.

# 

# Deep Network Tuning

## Deep Network Tuning

* DL is a programming language to extract information from data.

1. Some values will be filled by data later.
2. Differentiable.

* Various design patterns, from layers to network architecture

## Batch and layer Normalisation

### Batch Normalisation

* Standardising data makes the loss smoother for linear methods.

1. Smooth: .
2. A smaller  allows a larger learning rate.
3. Does not help deep NN.

* Batch Normalisation (BN) standards inputs for internal layers.

1. Improves the smoothness to make training easier.
2. (Still controversial why BN works.)

* Reshape input X into 2D (no change for 2D input )

1.   (batch n, channel c, width w, height h)

* Normalise by standardisation each column 

1. 

* Recovery  with  as the j-th column,  are parameters.
* Output **** by reshaping  to the same shape as .

### Batch Normalisation Code

**import** **tensorflow** **as** **tf**

**from** **d2l** **import** tensorflow **as** d2l

**def** batch\_norm(X, gamma, beta, moving\_mean, moving\_var, eps):

*# Compute reciprocal of square root of the moving variance elementwise*

inv = tf.cast(tf.math.rsqrt(moving\_var + eps), X.dtype)

*# Scale and shift*

inv \*= gamma

Y = X \* inv + (beta - moving\_mean \* inv)

**return** Y

Full code net: <http://d2l.ai/chapter_convolutional-modern/batch-norm.html>

### Layer Normalisation

* If apply to RNN, BN needs to maintain separated moving statistics for each time step.

1. Problematic for very long sequences during inference.

* Layer normalisation reshapes input  orrest is the same with BN.

1. Normalisation within each example, up to the current time step.
2. Consistent between training and inference.
3. Popularised by Transformers.

### More Normalizations

* Modify ‘reshape’, e.g.

1. InstanceNorm: n \* c \* w \* h  wh \* cn.
2. GroupNorm: n \* c \* w \* h  swh \* gn with c = sg.
3. CrossNorm: swap mean/std between a pair of features.

* Modify ‘normalise’: e.g. whitening.
* Modify ‘recovery’: e.g. replace  with a dense layer.
* Apply to weights or gradients.

### Summary

* Normalisation of inputs of internal layers makes deep NNs easier to train.
* A normalisation layer performs three steps: reshape input, normalise data, recovery with learnable parameters.

1. Notable examples include Batch Normalisation for CNNs, Layer Normalisation for Transformers.

# Transfer learning

## Transfer learning

* motivation

1. Exploit a model trained on one task for a related task.
2. Popular in deep learning as DNNs are data hungry and training cost is high.

* Approaches

1. Feature extraction (e.g. Word2Vec, ResNet-50 feature, I3D feature)
2. Train a model on a related task and reuse it.
3. Fine-tuning from a pertained model (focus of this lecture)

* Related to

1. Semi-supervised learning.
2. In the extreme, zero-shot / few-shot learning.
3. Multi-task learning, where some label data is available for earned.

### Transferring Knowledge

* There exists large-scale labelled CV datasets.

1. Especially for image classification, the cheapest one to label.

* Transfer knowledge from models trained on these datasets to your CV applications (with 10-100X smaller data).

### Pre-trained Models

* Partition a neural network into:

1. A feature extractor (encoder) maps raw pixels into linearly separable features.
2. A linear classifier (decode) makes decisions.

* Pre-trained model.

1. A neural network trained on a large-scale and general enough dataset.
2. The feature extractor may generalise well too.
3. Other datasets (e.g. medical/satellite images)
4. Other tasks (e.g. objection, segmentation)

### Fine-Tuning techniques

* Initialise the new model:

1. Initialise the feature extractor with the feature extractor parameters of a pre-trained model.
2. Randomly initialise the output layer.
3. Start the parameter optimization near a local minimal.

* Train with a small learning rate with just a few epochs.

1. Regularise the search space.

### Freeze Bottom Layers

* Neural networks learn hierarchical features.

1. Low-level features are universal, generalise well, e.g. curves / edges / blobs.
2. High-level features are more task and dataset specific, e.g. classification labels.

* Freeze bottom layers during fine tuning train the top layers from scratch.

1. Keep low-level universal features intact.
2. Focus on learning task specific features.
3. A strong regularizer.

### Where to Find Pre-trained Models

* Tensorflow Hub: https://tfhub.dev/

1. Tensorflow models submitted by users.

* TIMM: https://github.com/rwightman/pytorch-image-models

1. PyTorch models collected by Ross Wightman.
2. Code:

import timm

From torch import nn

model = timm.create\_model(‘resnet18’, pretrained=True)

model.fc = nn.Linear(model.fc.in\_features, n\_classes)

# Train model as a normal training job.

### Application

* Fine-tuning pre-trained models (on ImageNet) is widely used in various CV application.

1. Detection/segmentation (similar images but different targets)
2. Medical/satellite images (same task but very different images)

* Fine-tuning accelerates convergence.
* Though not always improve accuracy.

1. Training from scratch could get a similar accuracy, especially when the target dataset is also large.

### Summary

* Pre-train models on large-scale datasets (often image classification)
* Initialise weight with pre-trained models for down-stream tasks.
* Fine-tuning accelerates converges and (sometimes) improves accuracy.

## Fine-Tuning in NLP

### Self-supervised pre-training

* No large-scale labelled NLP dataset.
* Large quantities of unlabelled documents.

1. Wikipedia, ebooks, and crawled webpages.

* Self-supervised pre-training.

1. Generate “pseudo label” and use supervised learning tasks.
2. Common tasks for NLP.
3. Language model (LM): predict next word. e.g. I like your hat
4. Masked language model (MLM): random masked word prediction.

### Pre-trained Model

* Word embeddings: learn embeddings  and  for each word w

1. The masked word y can be predict by context words  via.



1. Embeddings u are used by other applications.

* Transformer based pre-trained models

1. BERT: a transformer **encoder** trained with masked words prediction and next sentence prediction.
2. GPT: a transformer **decoder** (will cover in prompt learning)
3. T5: a transformer **encoder-decoder** trained to fill space

#### BERT

Essay details: <https://www.bilibili.com/video/BV1PL411M7eQ>

* Pre-training tasks: masked token prediction, next sentence prediction.
* Pre-trained on Wikipedia and BookCorpus (>3B words)
* Many version: base / large, English / multilingual, cased / uncased
* Multiple variants

1. ALBERT
2. ELECTRA
3. ROBERTa

#### BERT Fine-tuning

* Randomly initialise the last layer, train a few epochs with small lr.
* Downstream task examples

### Practical Considerations

* BERT fine-tuning on small datasets can be unstable

1. BERT removed bias correction steps inAdam.
2. Too few (=3) epochs

* Randomly initialising some top transformer layers.

1. Features learned by top layers are too specific to the prediction tasks.
2. The cutoff depends on downstream tasks.

* More “tricks” later.

### Where to Find Pre-trained Model

* HuggingFace: a collection of pre-trained transformer models on both PyTorch and TensorFlow.

### Application

* “(BERT) obtains new state-of-the-art results on eleven natural language processing tasks”, including.

1. If a sequence of words is a grammatical English sentence
2. Sentiment of sentences from movie reviews
3. Sentences/questions in a pair are semantically equivalent, or similar.
4. If the premise entails the hypothesis
5. Find the span of the answer for a question.

* “(T5) achieve state-of-the-art results on many benchmarks covering summarization, question answering, text classification, and more.”

### Summary

* Self-supervised pre-training for NLP models.

1. A common task is (masked) language model.

* BERT is a giant transformer encoder.
* Downstream tasks fine-tune BERT with a consistent learner