Recap and Review – Exam 2

Applied Machine Learning
Derek Hoiem

Learning a model

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, Loss(f(\boldsymbol{X};\theta), \boldsymbol{y})$$

- $f(\boldsymbol{X};\theta)$: the model, e.g. $y = \boldsymbol{w}^T\boldsymbol{x}$
- θ: parameters of the model (e.g. $\boldsymbol{w}$)
- $(\boldsymbol{X}, \boldsymbol{y})$: pairs of training samples
- $Loss()$: defines what makes a good model
  - Good predictions, e.g. minimize $-\sum_n \log P(y_n|\boldsymbol{x}_n)$
  - Likely parameters, e.g. minimize $\boldsymbol{w}^T\boldsymbol{w}$
    - Regularization and priors indicate preference for particular solutions, which tends to improve generalization (for well chosen parameters) and can be necessary to obtain a unique solution

Prediction using a model

$$y_t = f(\boldsymbol{x_t}; \theta)$$

- Given some new set of input features $\boldsymbol{x}_t$, model predicts y_t
  - Regression: output y_t directly, possibly with some variance estimate
  - Classification
    - Output most likely y_t directly, as in nearest neighbor
    - Output $P(y_t|\boldsymbol{x}_t)$, as in logistic regression

Model evaluation process

1. Collect/define training, validation, and test sets
2. Decide on some candidate models and hyperparameters
3. For each candidate:
   a. Learn parameters with training set
   b. Evaluate trained model on the validation set
4. Select best model
5. Evaluate best model's performance on the test set
   - Cross-validation can be used as an alternative
   - Common measures include error or accuracy, root mean squared error, precision-recall

Bias-Variance Trade-off

$$\underbrace{E_{\mathbf{x},y,D}\left[(h_D(\mathbf{x}) - y)^2\right]}_{\text{Expected Test Error}} = \underbrace{E_{\mathbf{x},D}\left[(h_D(\mathbf{x}) - \bar{h}(\mathbf{x}))^2\right]}_{\text{Variance}} + \underbrace{E_{\mathbf{x},y}\left[(\bar{y}(\mathbf{x}) - y)^2\right]}_{\text{Noise}} + \underbrace{E_{\mathbf{x}}\left[(\bar{h}(\mathbf{x}) - \bar{y}(\mathbf{x}))^2\right]}_{\text{Bias}^2}$$

Variance: due to limited data
Different training samples will give different models that vary in predictions for the same test sample

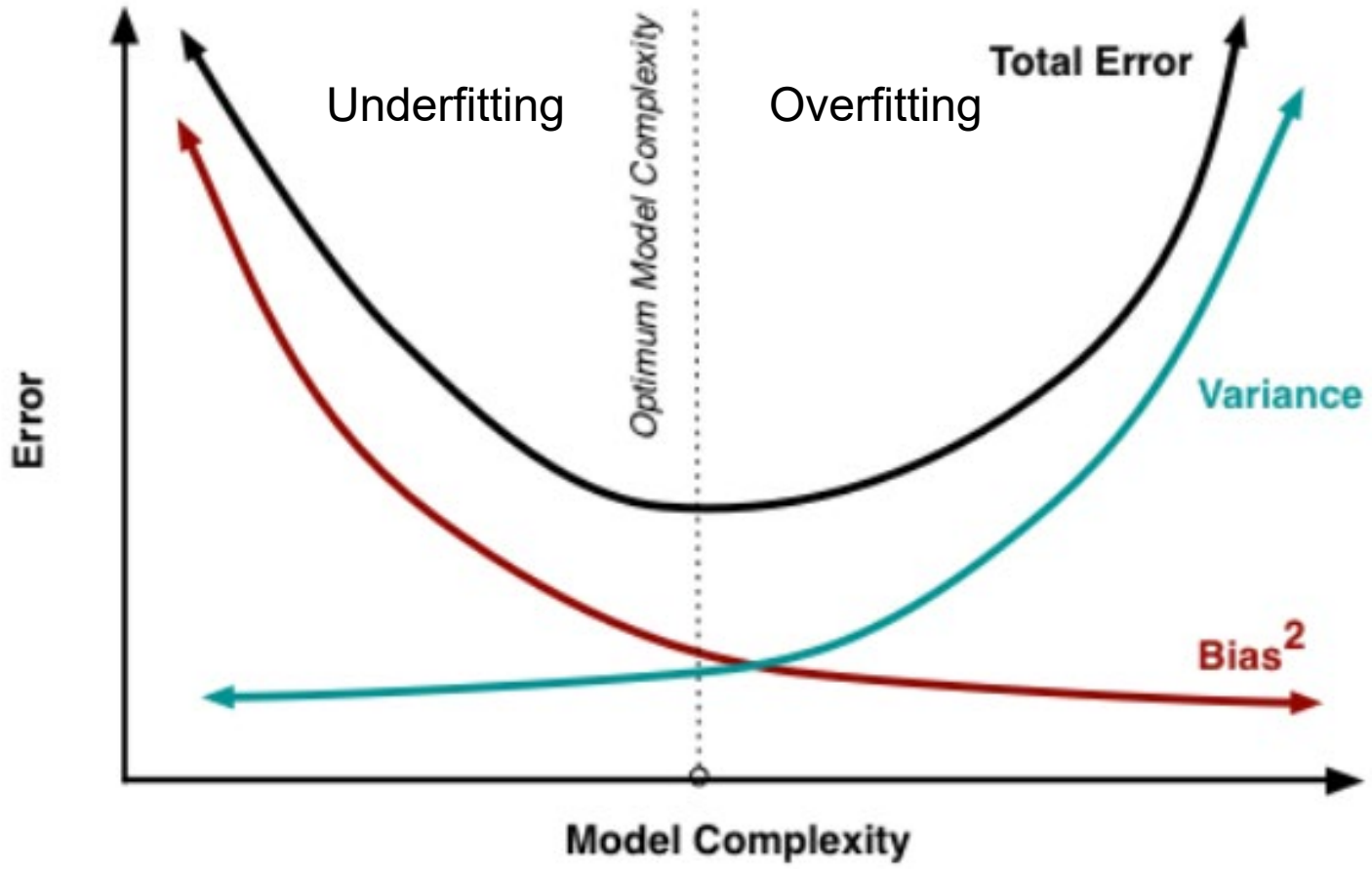
"**Noise**": irreducible error due to data/problem

Bias: error when optimal model is learned from infinite data

Above is for regression.
But same error = variance + noise + bias^2 holds for classification error and logistic regression.

See this for derivation

Fig Sources



How to detect high variance:

- Test error is much higher than training error

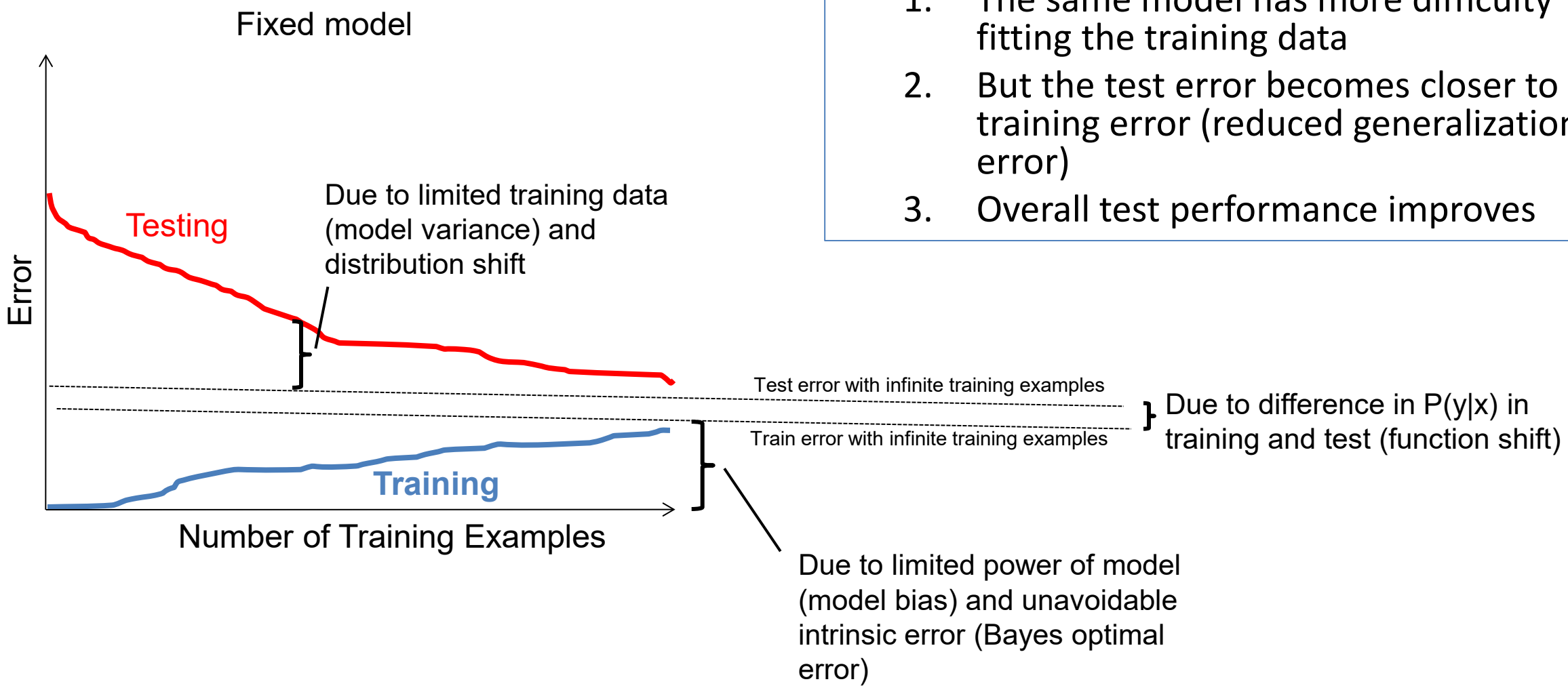
How to detect high bias or noise:

- The training error is high

As you increase model complexity:

- Training error will decrease
- Test error may decrease (if you are currently “underfitting”) or increase (if you are “overfitting”)

Performance vs training size



As we get more training data:

1. The same model has more difficulty fitting the training data
2. But the test error becomes closer to training error (reduced generalization error)
3. Overall test performance improves

Classification methods

	Nearest Neighbor	Naïve Bayes	Logistic Regression
Type	Instance-Based	Probabilistic	Probabilistic
Decision Boundary	Partition by example distance	Usually linear	Usually linear
Model / Prediction	$i^* = \underset{i}{\operatorname{argmin}}\, dist(X_{trn}[i], x)$ $y^* = y_{trn}[i^*]$	$y^* = \underset{y}{\operatorname{argmax}} \prod_i P(x_i \mid y) P(y)$	$w^T x + b \approx \log \frac{P(y=1 \mid x)}{P(y=0 \mid x)}$ $y^* = \underset{y}{\operatorname{argmax}}\, P(y \mid x)$
Strengths	* Low bias * No training time * Widely applicable * Simple	* Estimate from limited data * Simple * Fast training/prediction	* Powerful in high dimensions * Widely applicable * Good confidence estimates * Fast prediction
Limitations	* Relies on good input features * Slow prediction (in basic implementation)	* Limited modeling power	* Relies on good input features

Classification methods (extended)

assuming x in {0 1}

	Learning Objective	Training	Inference
Naïve Bayes	$\text{maximize} \sum_i \left[\sum_j \log P(x_{ij} \mid y_i; \theta_j) + \log P(y_i; \theta_0) \right]$	$\theta_{kj} = \frac{\sum_i \delta(x_{ij} = 1 \wedge y_i = k) + r}{\sum_i \delta(y_i = k) + Kr}$	$\theta_1^T \mathbf{x} + \theta_0^T (1 - \mathbf{x}) > 0$ where $\theta_{1j} = \log \frac{P(x_j = 1 \mid y = 1)}{P(x_j = 1 \mid y = 0)}$, $\theta_{0j} = \log \frac{P(x_j = 0 \mid y = 1)}{P(x_j = 0 \mid y = 0)}$
Logistic Regression	$\text{minimize} \sum_i -\log(P(y_i \mid \mathbf{x}, \boldsymbol{\theta})) + \lambda \lVert \boldsymbol{\theta} \rVert$ where $P(y_i \mid \mathbf{x}, \boldsymbol{\theta}) = 1/(1 + \exp(-y_i \boldsymbol{\theta}^T \mathbf{x}))$	Gradient descent	$\boldsymbol{\theta}^T \mathbf{x} > t$
Linear SVM	$\text{minimize } \lambda \sum_i \xi_i + \frac{1}{2} \lVert \boldsymbol{\theta} \rVert$ such that $y_i \boldsymbol{\theta}^T \mathbf{x} \geq 1 - \xi_i \ \ \forall i, \ \xi_i \geq 0$	Quadratic programming or subgradient opt.	$\boldsymbol{\theta}^T \mathbf{x} > t$
Kernelized SVM	complicated to write	Quadratic programming	$\sum_i y_i \alpha_i K(\hat{\mathbf{x}}_i, \mathbf{x}) > 0$
Nearest Neighbor	most similar features → same label	Record data	y_i where $i = \underset{i}{\text{argmin}} \ K(\hat{\mathbf{x}}_i, \mathbf{x})$

* Notation may differ from previous slide

Regression methods

	Nearest Neighbor	Naïve Bayes	Linear Regression
Type	Instance-Based	Probabilistic	Data fit
Decision Boundary	Partition by example distance	Usually linear	Linear
Model / Prediction	$i^* = \underset{i}{\operatorname{argmin}}\, dist(X_{trn}[i], x)$ $y^* = y_{trn}[i^*]$	$y^* = \underset{y}{\operatorname{argmax}} \prod_i P(x_i \mid y) P(y)$	$y^* = w^T x$
Strengths	* Low bias * No training time * Widely applicable * Simple	* Estimate from limited data * Simple * Fast training/prediction	* Powerful in high dimensions * Widely applicable * Fast prediction * Coefficients may be interpretable
Limitations	* Relies on good input features * Slow prediction (in basic implementation)	* Limited modeling power	* Relies on good input features

Entropy and Information Gain

- Entropy, $H(X)$: measures uncertainty of X
- Specific conditional entropy, $H(X|Y = y)$: measures uncertainty of X if Y is known to have a particular value
- Conditional entropy $H(X|Y)$: measures expected uncertainty of X if I know Y
- Information gain $I(X|Y)$: measures how much knowing Y would reduce my uncertainty in X

$$H(X) = -\sum_{x} P(X = x) \log_2 P(X = x)$$

$$H(X|Y = y) = -\sum_{x} P(X = x|Y = y) \log_2 P(X = x|Y = y)$$

$$H(X|Y) = -\sum_{y} H(X|Y = y) P(Y = y)$$

$$\mathrm{I}(X|Y) = H(X) - H(X|Y)$$

Deep Learning

Pegasos with **mini-batch**

- Calculating gradient based on multiple examples reduces variance of gradient estimate

INPUT: S, λ, T, k
INITIALIZE: Set $\mathbf{w}_1 = 0$
FOR $t = 1, 2, \ldots, T$
Choose $A_t \subseteq [m]$, where $|A_t| = k$, uniformly at random
Set $A_t^+ = \{i \in A_t : y_i \langle \mathbf{w}_t, \mathbf{x}_i \rangle < 1\}$
Set $\eta_t = \frac{1}{\lambda t}$
Set $\mathbf{w}_{t+1} \leftarrow (1 - \eta_t \lambda)\mathbf{w}_t + \frac{\eta_t}{k} \sum_{i \in A_t^+} y_i \mathbf{x}_i$

OUTPUT: $\mathbf{w}_{T+1}$

k: batch size
m: number of training samples
A_t: batch of examples
A_t^+: examples within margin

S: training set
λ: regularization weight
T: number iterations
$\boldsymbol{w}_t$: model weights
$\boldsymbol{x}_i$: features for example i
y_i: label for example i
η_t: step size (“learning rate”)

Adam: Adaptive Moment Estimation

Adam:

$m_t = \beta \cdot m_t + (1 - \beta) \cdot g(w_t)$ [momentum, $\beta = 0.9$]

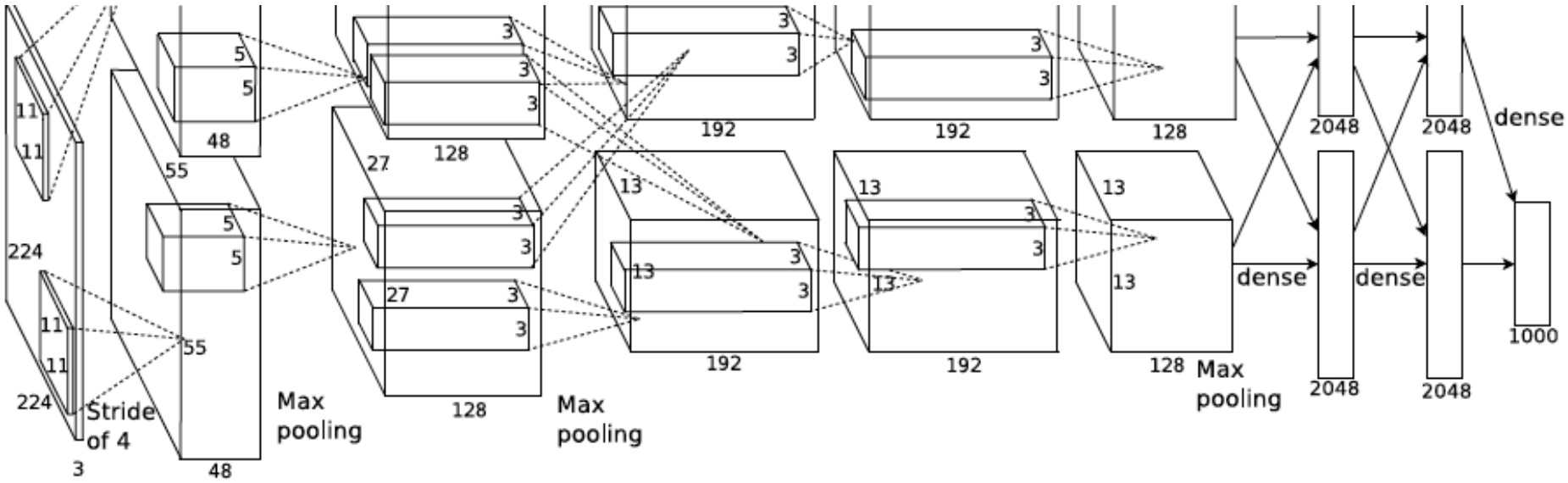
$g_{sq}(t) = \epsilon \cdot g_{sq}(t-1) + (1 - \epsilon) \cdot g(w_t)^2$ [RMSProp, $\epsilon = 0.999$]

$\Delta w_t = -\eta \cdot m_t / \sqrt{g_{sq}(w_t)}$

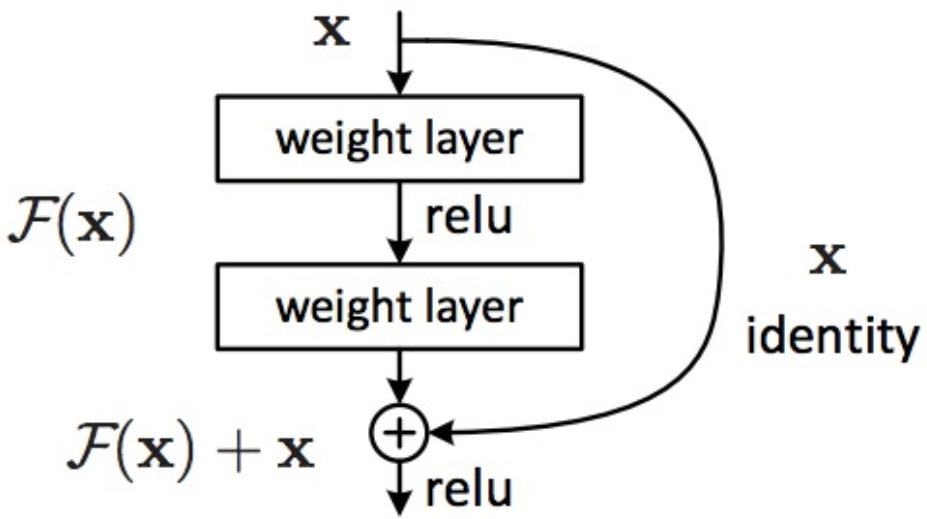
$w_{t+1} = w_t + \Delta w_t$

AdamW is widely used and easier to tune than SGD + momentum

AlexNet CNN


224
224
3
11
11
11
11
Stride
of 4
55
55
5
5
48
48
Max
pooling
27
27
5
5
128
128
3
3
3
3
3
3
Max
pooling
13
13
13
13
13
13
192
192
192
192
128
128
Max
pooling
dense
dense
2048
2048
2048
2048
dense
1000

ResNet Block



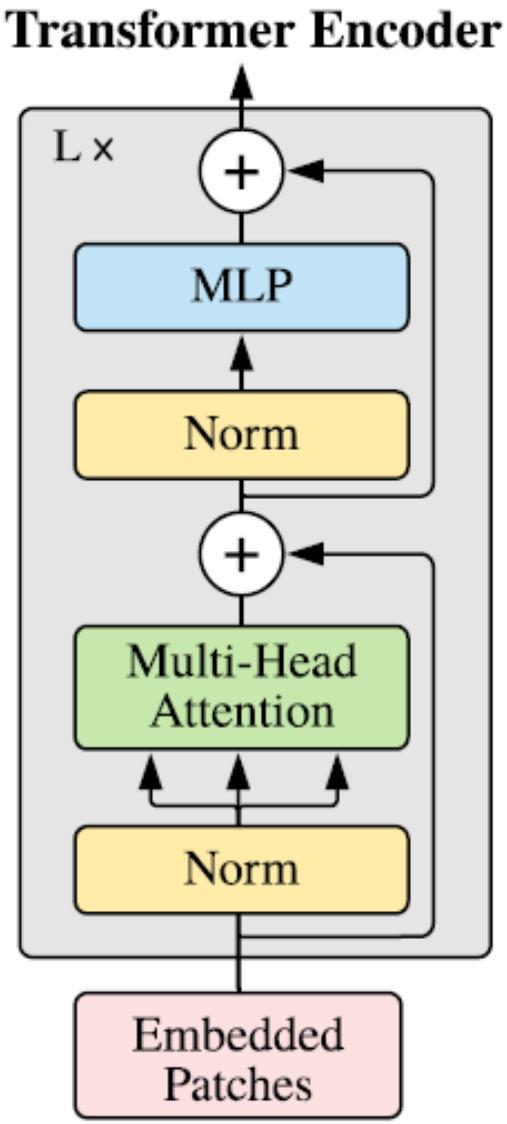
ResNet - 18

```
class Network(nn.Module):
    def __init__(self, num_classes=1000):
        super().__init__()
        resblock = ResBlock
        self.layer0 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU()
        )
        self.layer1 = nn.Sequential(
            resblock(64, 64, downsample=False),
            resblock(64, 64, downsample=False)
        )
        self.layer2 = nn.Sequential(
            resblock(64, 128, downsample=True),
            resblock(128, 128, downsample=False)
        )
        self.layer3 = nn.Sequential(
            resblock(128, 256, downsample=True),
            resblock(256, 256, downsample=False)
        )
        self.layer4 = nn.Sequential(
            resblock(256, 512, downsample=True),
            resblock(512, 512, downsample=False)
        )
        self.gap = torch.nn.AdaptiveAvgPool2d(1)
        self.fc = torch.nn.Linear(512, num_classes)
```

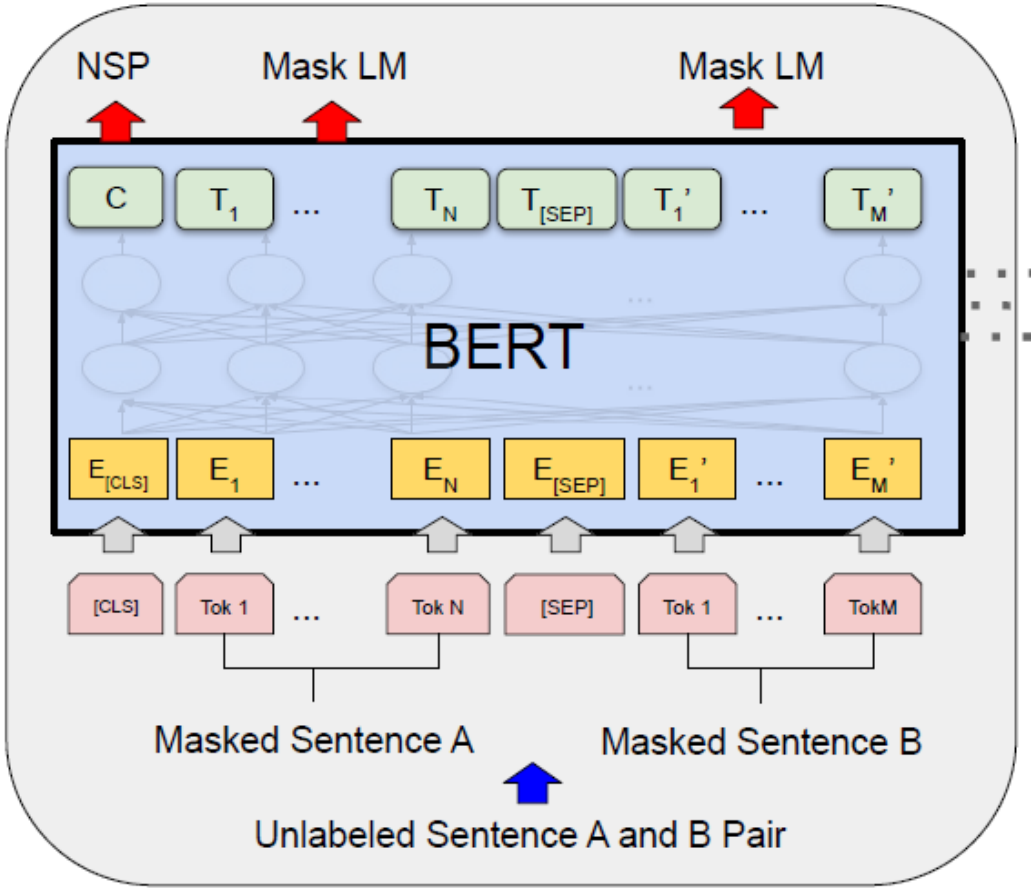
```
def forward(self, input):
    input = self.layer0(input)
    input = self.layer1(input)
    input = self.layer2(input)
    input = self.layer3(input)
    input = self.layer4(input)
    input = self.gap(input)
    input = torch.flatten(input, 1)
    input = self.fc(input)

    return input
```

Transformer Block



BERT

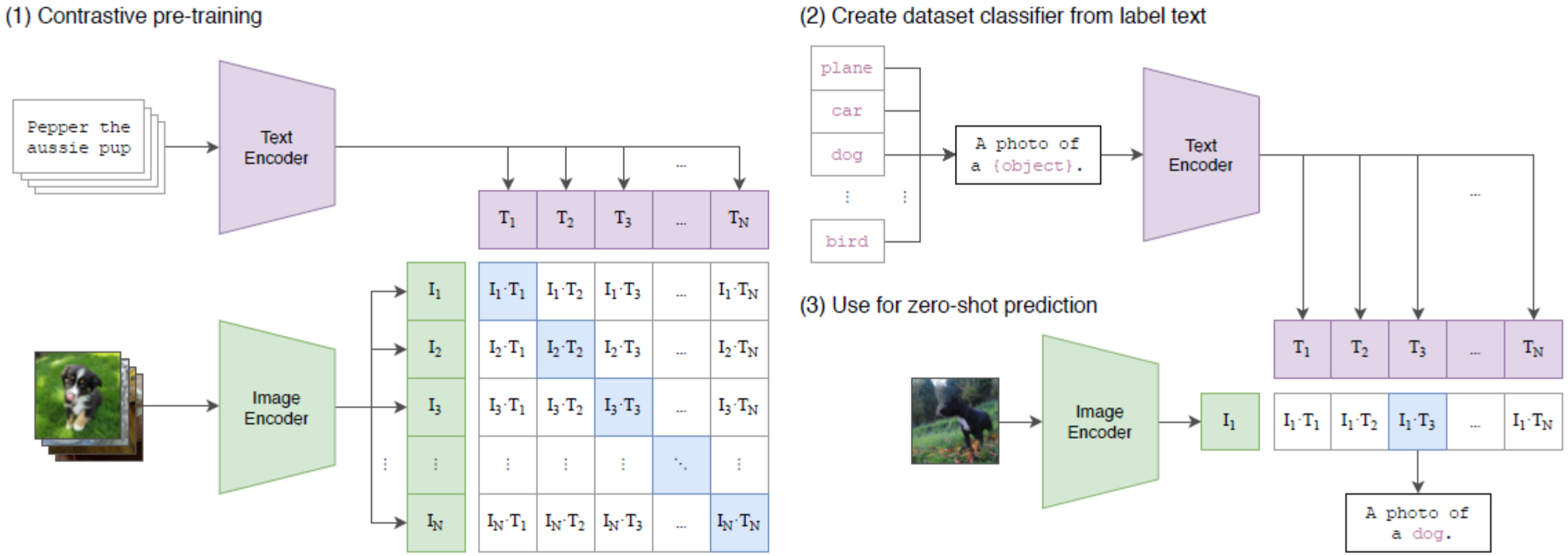


Pre-training

GPT Series

- All Transformer-based, similar to BERT
- GPT: generative-pretraining (GPT) is effective for large language models
  - Learns to predict the next word given preceding words
- GPT-2: GPT models can perform reasonable zero-shot task performance with larger models trained on more data
- GPT-3: Even larger GPT models trained on even more data are good at many tasks, especially text generation, and can be “trained” at inference time with in-context examples

CLIP: Learning Transferrable Models from Natural Language Supervision (Radford et al. 2021)



How to apply linear probe

Pre-compute features method

1. Load pretrained model (many available)
   https://pytorch.org/vision/stable/models.html
2. Remove prediction final layer
3. Apply model to each image to get features; save them with labels
4. Train new linear model (e.g. logistic regression or SVM) on the features

```
import torch
import torch.nn as nn
from torchvision import models

model = models.alexnet(pretrained=True)

# remove last fully-connected layer
new_classifier = nn.Sequential(*list(model.classifier.children())[:-1])
model.classifier = new_classifier
```

Freeze encoder method

1. Load pretrained model (many available)
   https://pytorch.org/vision/stable/models.html
2. Set network to not update weights
3. Replace last layer
4. Retrain network with new dataset

- Slower than method on left but does not require storing features, and can apply data augmentation

```
model = torchvision.models.vgg19(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
    # Replace the last fully-connected layer
    # Parameters of newly constructed modules have requires_grad=True by default
model.fc = nn.Linear(512, 8) # assuming that the fc7 layer has 512 neurons, other
model.cuda()
```

Source

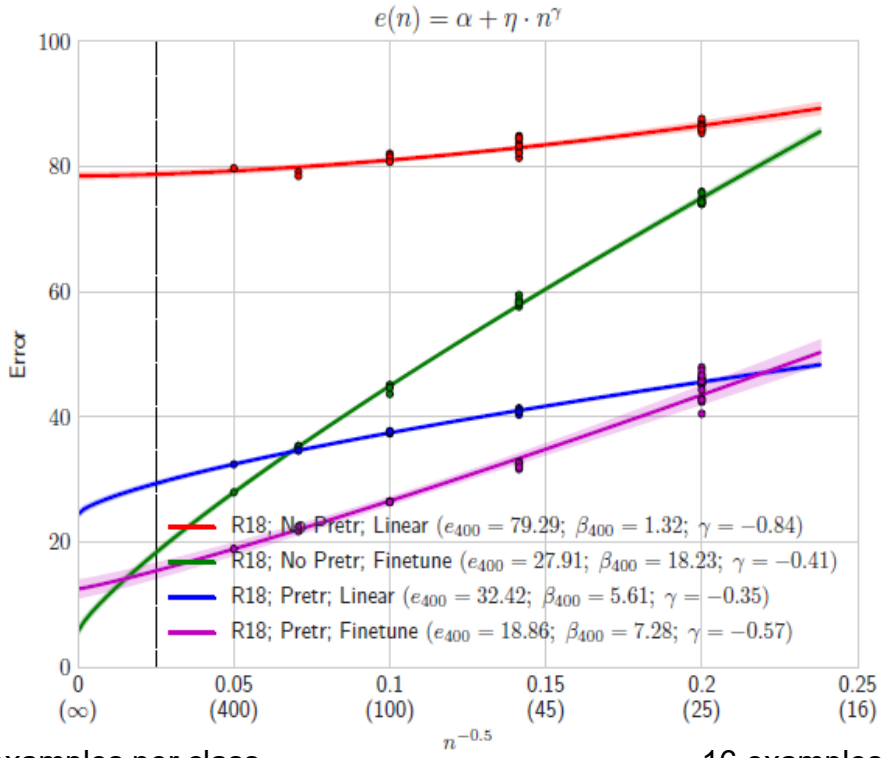
How to apply fine-tuning

1. Load pre-trained model
2. Replace last layer
3. Set a low learning rate (e.g. lr=e-4)
   - Very sensitive to learning rate because you want to improve but not drift too far from the initial model -- learning rate is the most critical parameter for fine-tuning!
   - Learning rate is often at least 10x lower than from “scratch” training
   - Can “warm start” by freezing earlier layers initially and then unfreezing after a few epochs when the linear layer is mostly trained (avoids messing up encoder while classifier is adjusting), or start learning rate near zero and increase slowly over several epochs
   - Can set lower learning rate for earlier layers

```
target_class = 37
model = torch.hub.load('pytorch/vision:v0.10.0', 'resnet34', pretrained=True)
model.fc = nn.Linear(512, target_class)
```

In this example, last layer has 512 input features and is called “fc”

Comparing linear probe, fine-tuning, and training from scratch, when does each have an advantage and why?



400+ examples per class

16 examples per class

(a) Transfer: ImageNet to Cifar100

"Learning Curves" (2021) pdf

ResNet18, Err vs # examples / class (in paren)

Green: Train from scratch
Blue: Linear Probe from ImageNet
Purple: Fine-tune from ImageNet

Very little data
- Use linear probe on pre-trained model

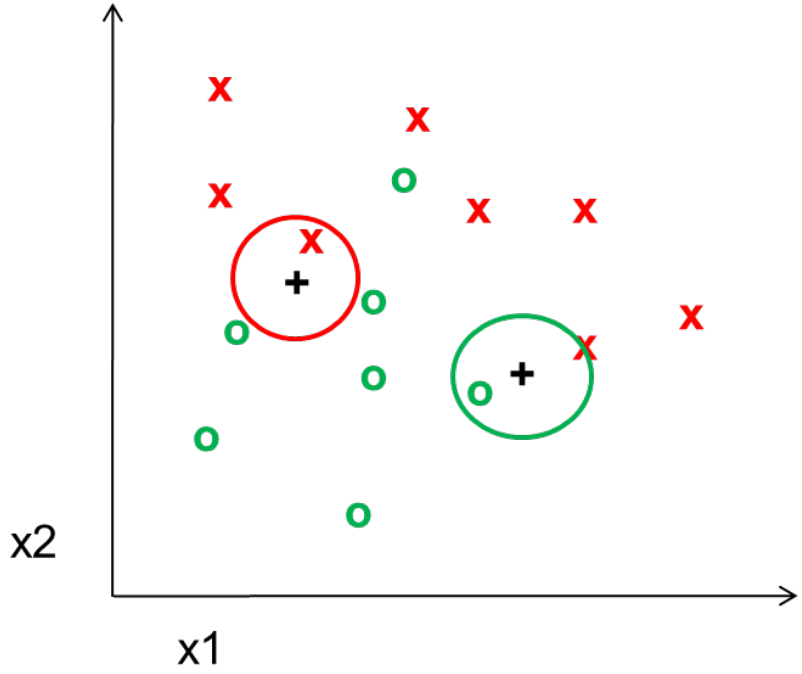
Moderate data
- Fine-tune pre-trained model

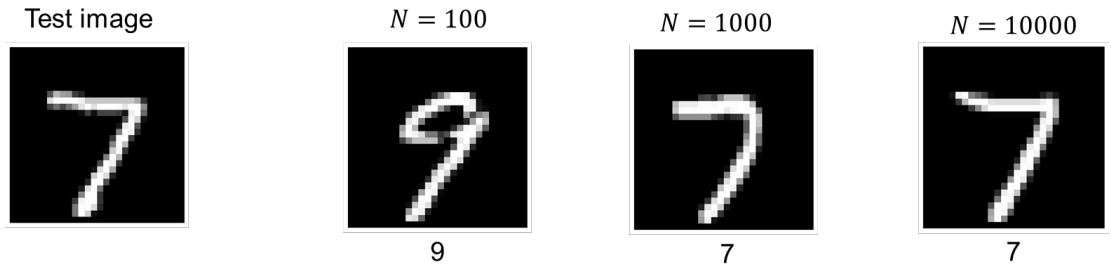
Very large dataset
- Either fine-tune or train from scratch

Summaries

KNN Classification (L2)

- Foundation of ML: similar features predict similar labels
  - Hard part: How to represent inputs with vectors that reflect the similarity
- KNN is a simple but effective classifier that predicts the label of the most similar training example(s)
  - Accuracy depends on quality of features and number of training samples
- Larger K gives a smoother prediction function
- Measure classification performance with error and confusion matrices





Working with Data (L2)

- **Data** is a set of numbers that contains information. Images, audio, signals, tabular data and everything else must be represented as a vector of numbers to be used in ML.

- **Information** is the power to predict something – a lot of the challenge in ML is in transforming the data to make the desired information more obvious

- In machine learning, we have
  Sample: a data point, such as a feature vector and label corresponding to the input and desired output of the model
  Dataset: a collection of samples
  Training set: a dataset used to train the model
  Validation set: a dataset used to select which model to use or compare variants and manually set parameters
  Test set: a dataset used to evaluate the final model

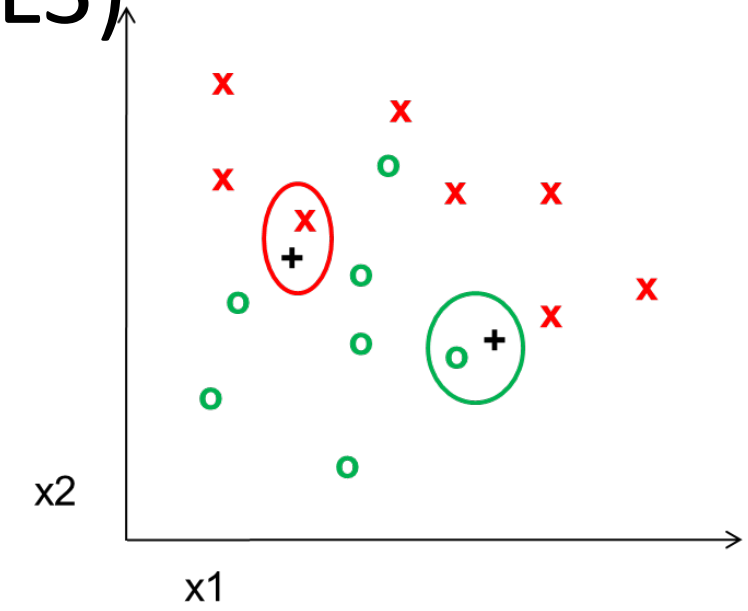
- In a **classification** problem, the goal is to map from features to a categorical label (or “class”)

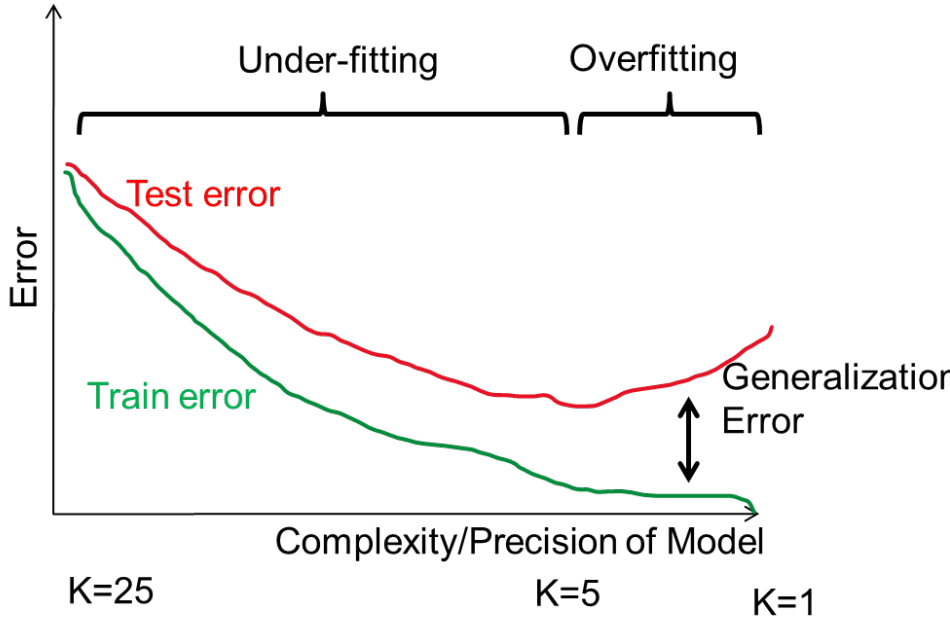
- Nearest neighbor (or **K-NN**) algorithm can perform classification by retrieving the K nearest neighbors to the query features and assigning their most common label

- We can measure **error** and **confusion matrices** to show the fraction of mistakes and what kinds of mistakes are made

KNN Regression & Generalization (L3)

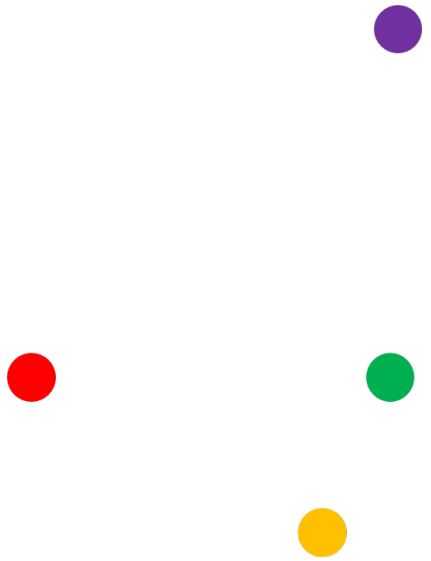
- Similarity/distance measures: L1, L2, cosine
- KNN can be used for either classification (return most common label) or regression (return average target value)
- Regression error measures
  - Root mean squared error(RMSE)
    $\sqrt{\frac{1}{N}\sum_i (f(X_i) - y_i)^2}$
  - R^2: $1 - \frac{\sum_i (f(X_i) - y_i)^2}{\sum_i (y_i - \bar{y})^2}$
- Test error is composed of
  - **Irreducible error** (perfect prediction not possible given features)
  - **Bias** (model cannot perfectly fit the true function)
  - **Variance** (parameters cannot be perfectly learned from training data)

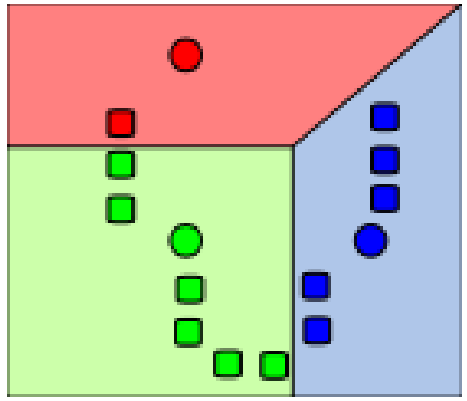




Clustering (L4)

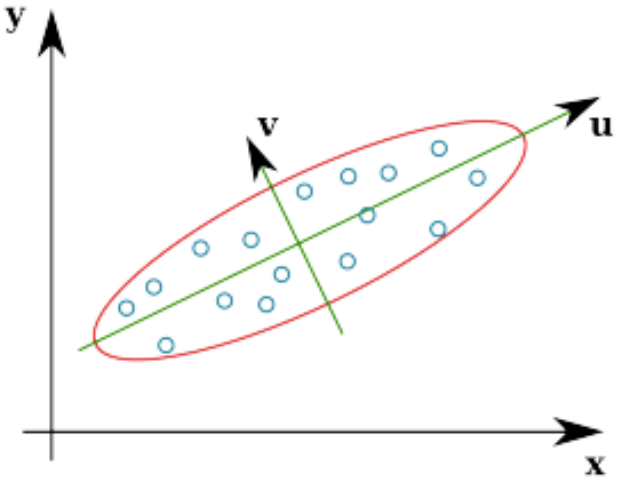
- Use highly optimized libraries like FAISS for search/retrieval
- Approximate search methods like LSH can be used to find similar points quickly
- Clustering groups similar data points
- K-means is the must-know method, but there are many others



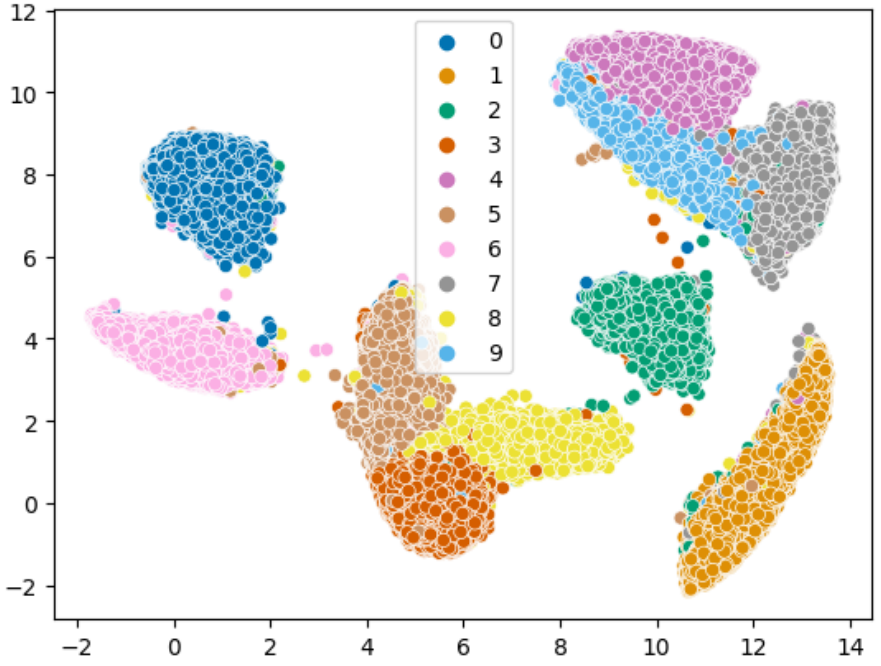


PCA/Embedding (L5)

- PCA reduces dimensions by linear projection
  - Preserves variance to reproduce data as well as possible, according to mean squared error
  - May not preserve local connectivity structure or discriminative information



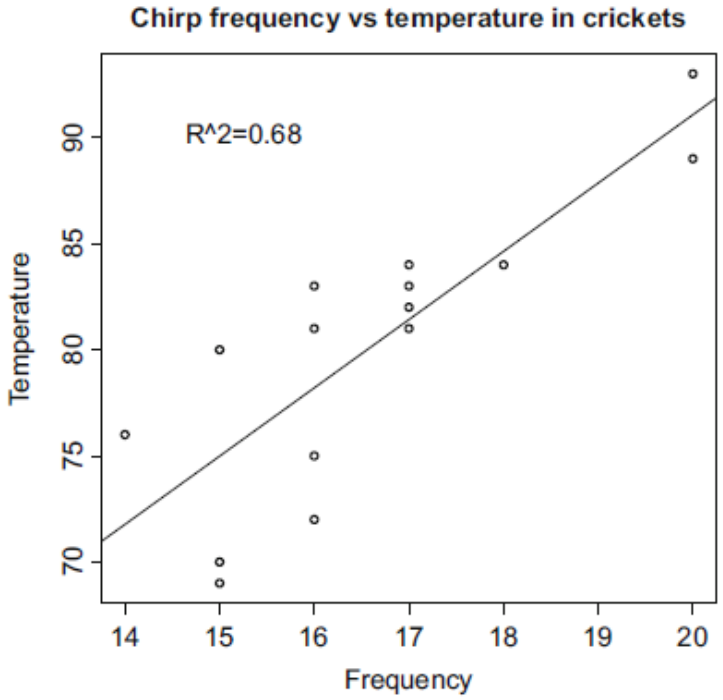
- Other methods try to preserve relationships between points
  - MDS: preserve pairwise distances
  - IsoMap: MDS but using a graph-based distance
  - t-SNE: preserve a probabilistic distribution of neighbors for each point (also focusing on closest points)
  - UMAP: incorporates k-nn structure, spectral embedding, and more to achieve good embeddings relatively quickly

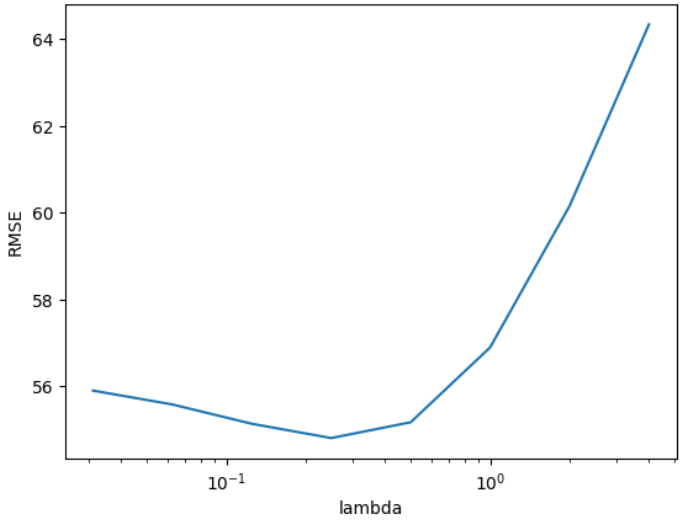


Linear Regression (L6)

- Linear regression fits a linear model to a set of feature points to predict a continuous value
  - Explain relationships
  - Predict values
  - Extrapolate observations

- Regularization prevents overfitting by restricting the magnitude of feature weights
  - L1: prefers to assign a lot of weight to the most useful features
  - L2: prefers to assign smaller weight to everything





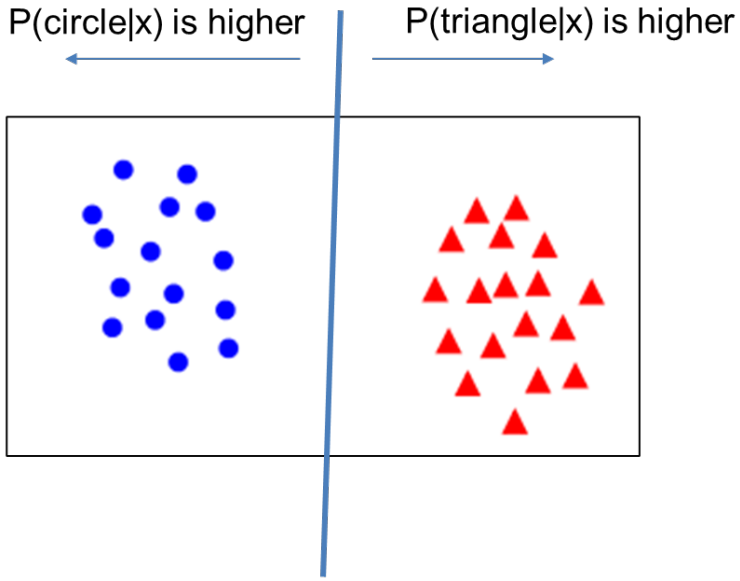
Linear Classifiers (L7)

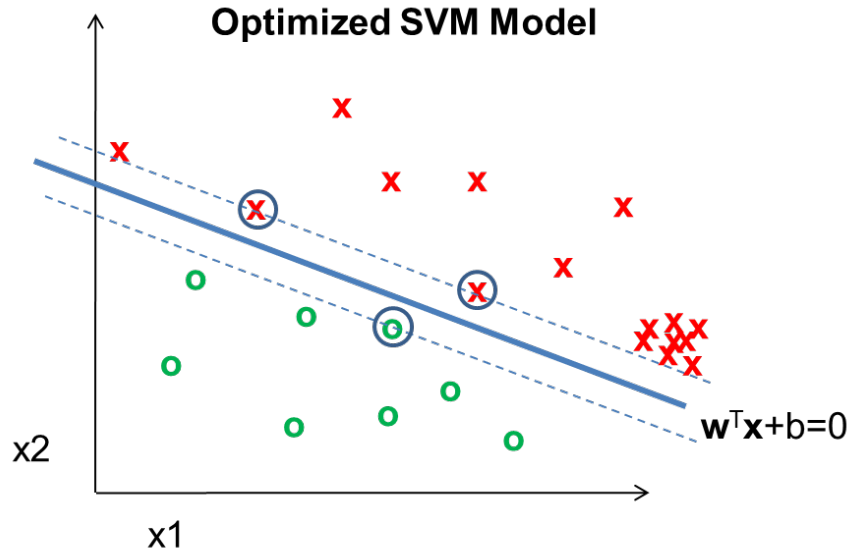
- Linear logistic regression and linear SVM are classification techniques that aims to split features between two classes with a linear model
  - Predict categorical values with confidence

- Logistic regression maximizes confidence in the correct label, while SVM just tries to be confident enough

- Non-linear versions of SVMs can also work well and were once popular (but almost entirely replaced by deep networks)

- Nearest neighbor and linear models are the final predictors of most ML algorithms – the complexity lies in finding features that work well with NN or linear models





Probability / Naïve Bayes (L8)

- Probabilistic models are a large class of machine learning methods

- Naïve Bayes assumes that features are independent given the label
  - Easy/fast to estimate parameters
  - Less risk of overfitting when data is limited

$$P(\boldsymbol{x}, y) = \prod_i P(x_i|y)P(y)$$

- You can look up how to estimate parameters for most common probability models
  - Or take partial derivative of total data/label likelihood given parameter

- Prediction involves finding y that maximizes $P(x, y)$, either by trying all y or solving partial derivative

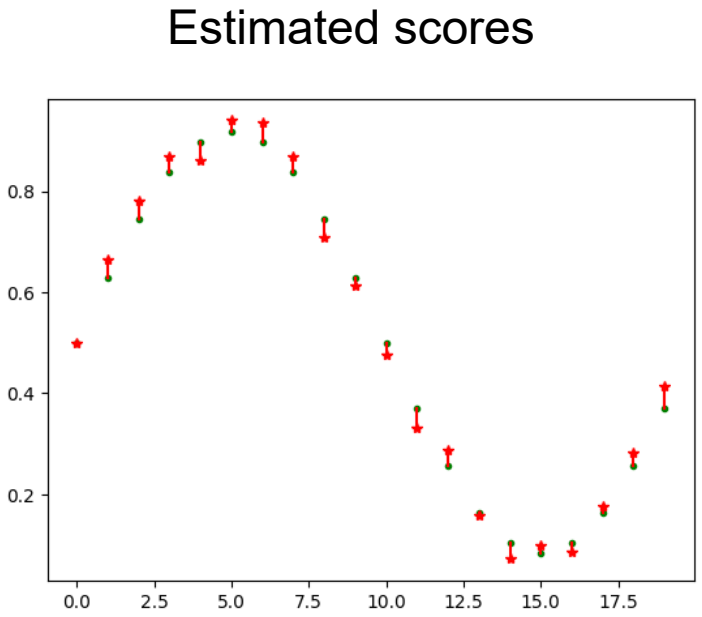
- Maximizing $\log P(x, y)$ is equivalent to maximizing $P(x, y)$ and often much easier

$$y^* = \operatorname{argmax}_y \prod_i P(x_i|y)\,P(y)$$

$$= \operatorname{argmax}_y \sum_i \log P(x_i|y) + \log P(y)$$

EM (L9)

- EM is a widely applicable algorithm to solve for latent variables and parameters that make the observed data likely
  - E-step: compute the likelihoods of the values of the latent variables
  - M-step: solve for most likely model parameters, using the likelihoods from the E-step as weights
- While derivation is long and somewhat complicated, the application is simple
- EM is used, for example, in mixture of Gaussian and topic models



Good annotators: 0, 1, 3

PDF Estimation (L10)

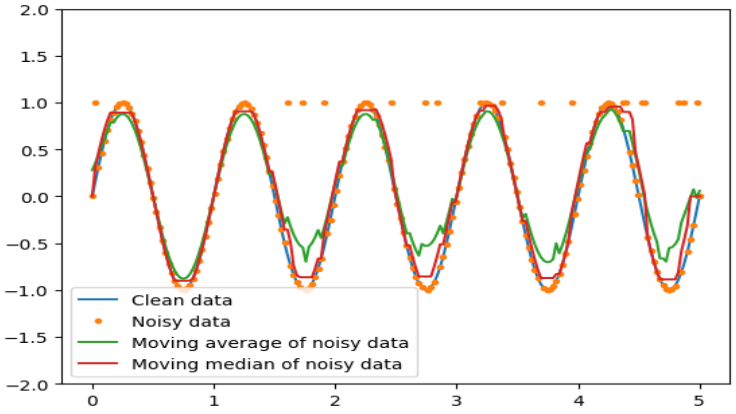
	Parametric Models	**Semi-Parametric**	**Non-Parametric**
Description	Assumes a fixed form for density	Can fit a broad range of functions with limited parameters	Can fit any distribution
Examples	Gaussian, exponential	Mixture of Gaussians	Discretization, kernel density estimation
Good when	Model is able to approximately fit the distribution	Low dimensional or smooth distribution	1-D data
Not good when	Model cannot approximate the distribution	Distribution is not smooth, challenging in high dimensions	Data is high dimensional

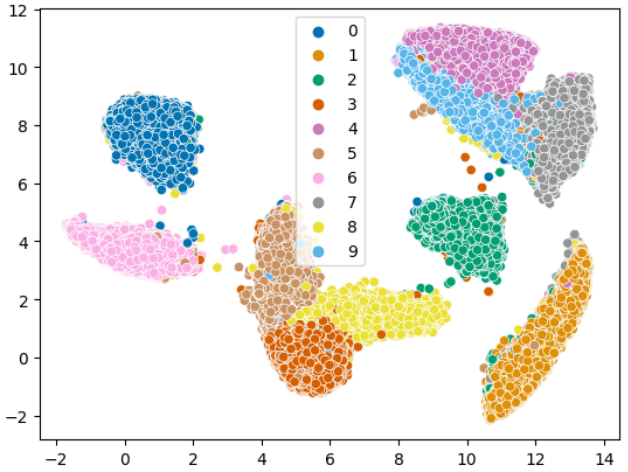
Robust Estimation (L11)

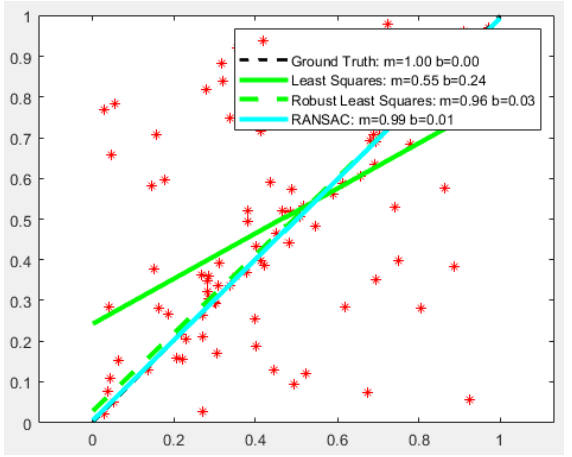
Median and quantiles are robust to outliers, while mean/min/max aren't

Outliers can be detected as low probability points, low density points, poorly compressible points, or through 2D visualizations

Least squares is not robust to outliers. Use RANSAC or IRLS or robust loss function instead.

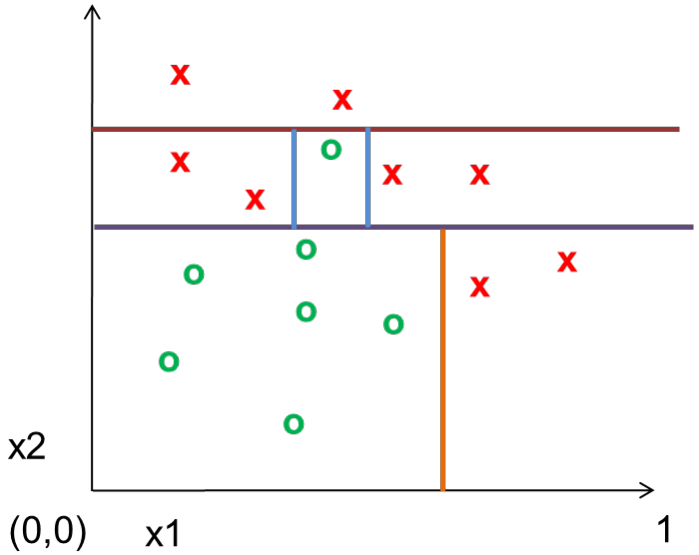






Decision Trees (L12)

- Decision/regression trees learn to split up the feature space into partitions with similar values



- Entropy is a measure of uncertainty

$$H(X) = -\sum_{x \in X} p(x) \log_2 p(x)$$

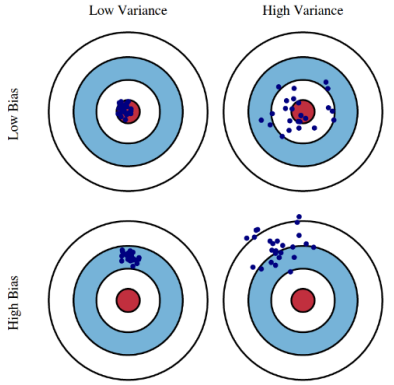
- Information gain measures how much particular knowledge reduces prediction uncertainty

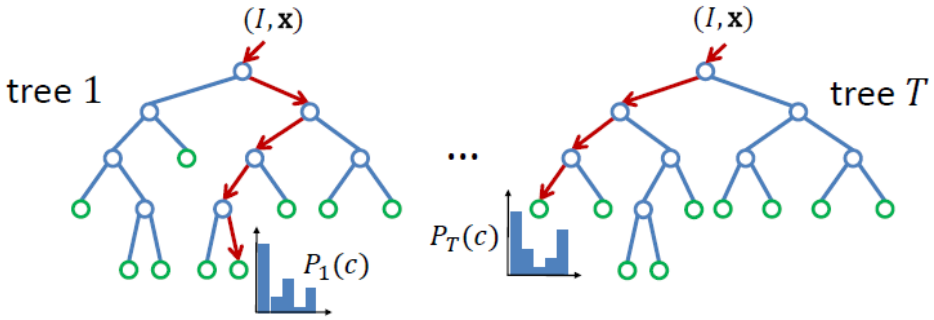
$$IG(Y|X) \quad = \quad H(Y) - H(Y|X)$$

Ensembles and Forests (L13)

- Ensembles improve accuracy and confidence estimates by reducing bias and/or variance
- Boosted trees minimize bias by fixing previous mistakes
- Random forests minimize variance by averaging over multiple different trees
- Random forests and boosted trees are powerful classifiers and useful for a wide variety of problems

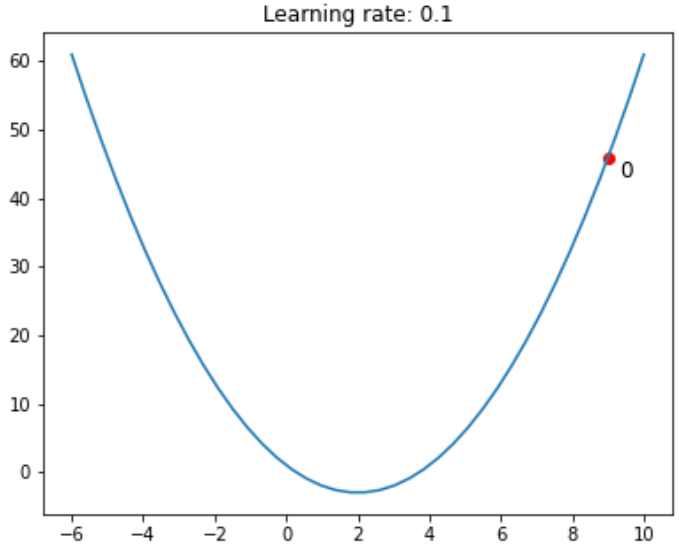
$$\underbrace{E_{\mathbf{x},y,D}\left[(h_D(\mathbf{x})-y)^2\right]}_{\text{Expected Test Error}} = \underbrace{E_{\mathbf{x},D}\left[(h_D(\mathbf{x})-\bar{h}(\mathbf{x}))^2\right]}_{\text{Variance}} + \underbrace{E_{\mathbf{x},y}\left[(\bar{y}(\mathbf{x})-y)^2\right]}_{\text{Noise}} + \underbrace{E_{\mathbf{x}}\left[(\bar{h}(\mathbf{x})-\bar{y}(\mathbf{x}))^2\right]}_{\text{Bias}^2}$$

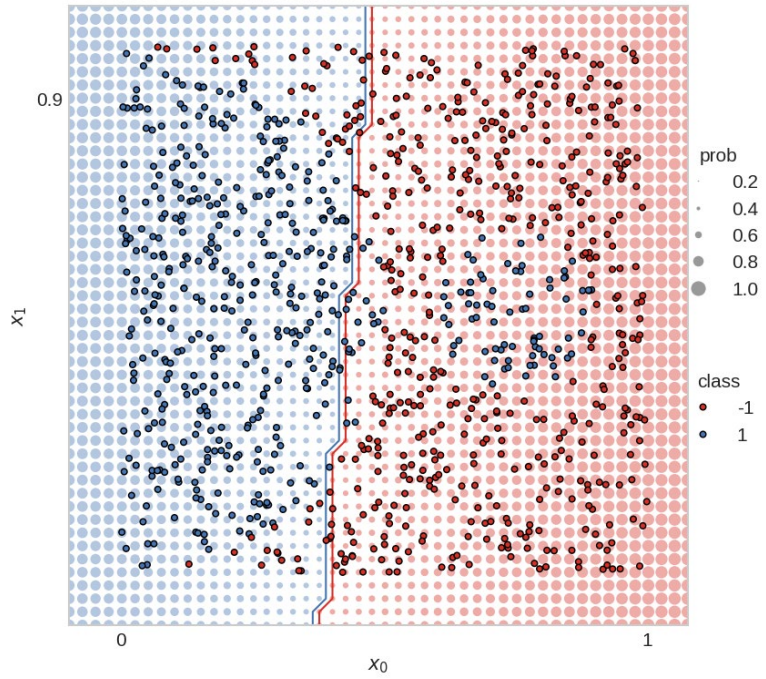




SGD (L14)

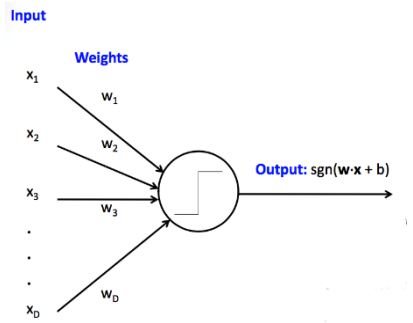
- Gradient descent iteratively steps in direction of negative gradient of loss
- Stochastic gradient descent estimates gradient using small batches of samples
  - Faster than full gradient descent
- Linear models have limited ability to fit the data – often need non-linear models like multilayer networks

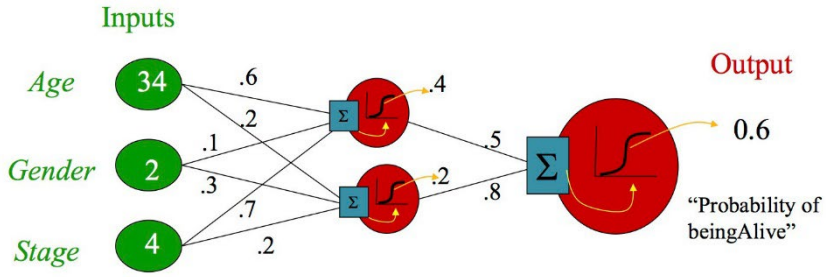




MLPs and Backprop (L15)

- Perceptrons are linear prediction models
- MLPs are non-linear prediction models, composed of multiple linear layers with non-linear activations
- MLPs can model more complex functions, but are harder to optimize
- Optimization is by a form of stochastic gradient descent
- Deeper networks are subject to vanishing gradient problems that are reduced (but not eliminated) with ReLU activations





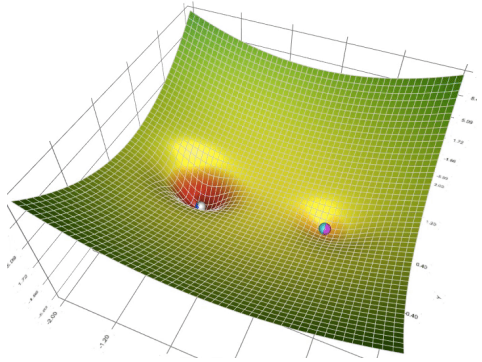
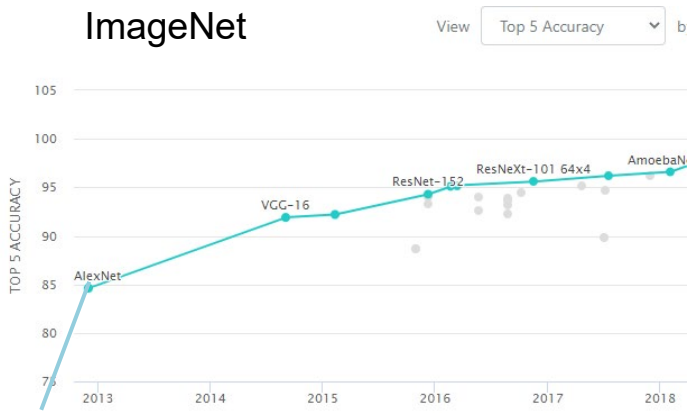
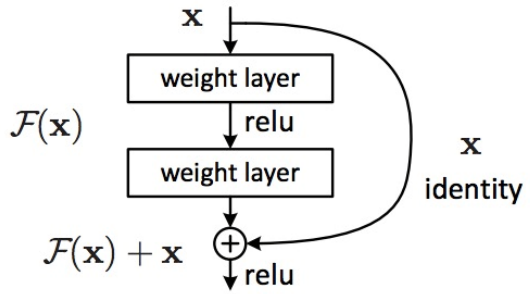


Figure source

CNNs Keys to Deep Learning (L16)

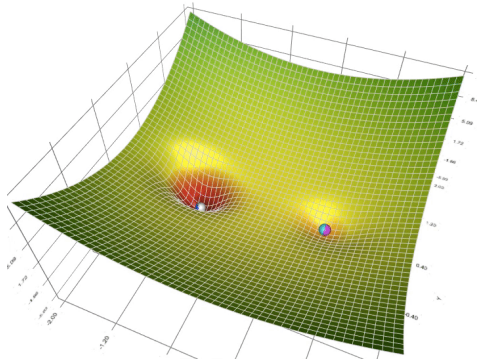
- Deep networks provide huge gains in performance
  - Large capacity, optimizable models
  - Learn from new large datasets





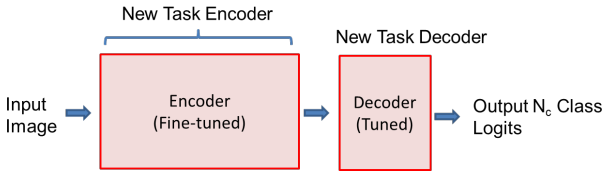
- ReLU and skip connections simplify optimization

- SGD+momentum and AdamW are the most commonly used optimizers

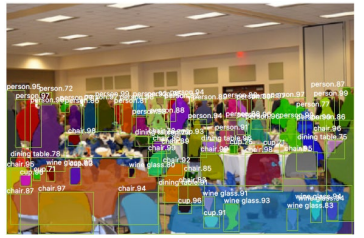


Deep Learning Optimization and Computer Vision (L17)

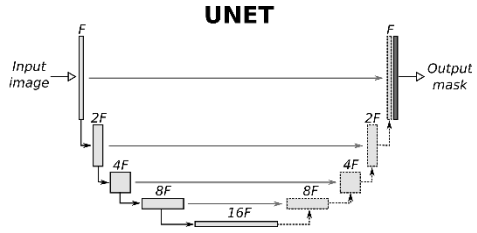
- Models trained on ImageNet are used as pretrained “backbones” for other vision tasks



- Mask-RCNN samples patches in feature maps and predicts boxes, object region, and keypoints



- Many image generation and segmentation methods are based on U-Net downsamples while deepening features, then upsamples with skip connections



Words and Attention (L18)

Sub-word tokenization based on byte-pair encoding is an effective way to turn natural text into a sequence of integers

Chair is broken →
ch##, ##air, is, brok##, ##en

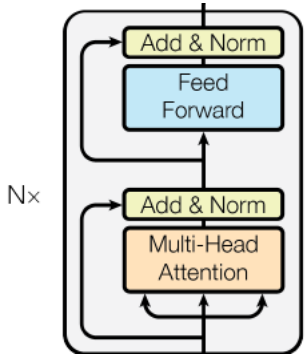
Learned vector embeddings of these integers model the relationships between words

Paris – France
+ Italy = Rome

Attention is a general processing mechanism that regresses or clusters values

Input (k,q,v)	iter 1	iter 2	iter 3	iter 4
1.000	1.497	1.818	1.988	2.147
9.000	8.503	8.182	8.012	7.853
8.000	8.128	8.141	8.010	7.853
2.000	1.872	1.859	1.990	2.147

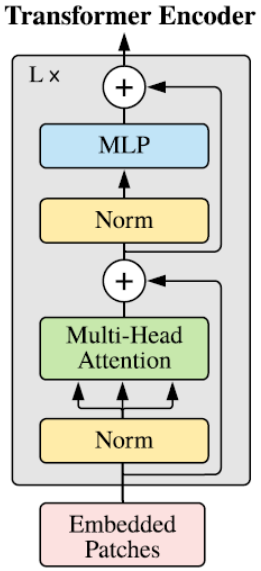
Stacked transformer blocks are a powerful network architecture that alternates attention and MLPs

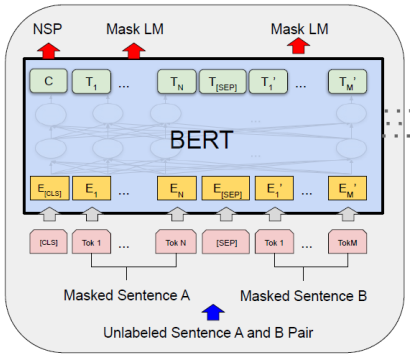


Further reading: http://nlp.seas.harvard.edu/annotated-transformer/

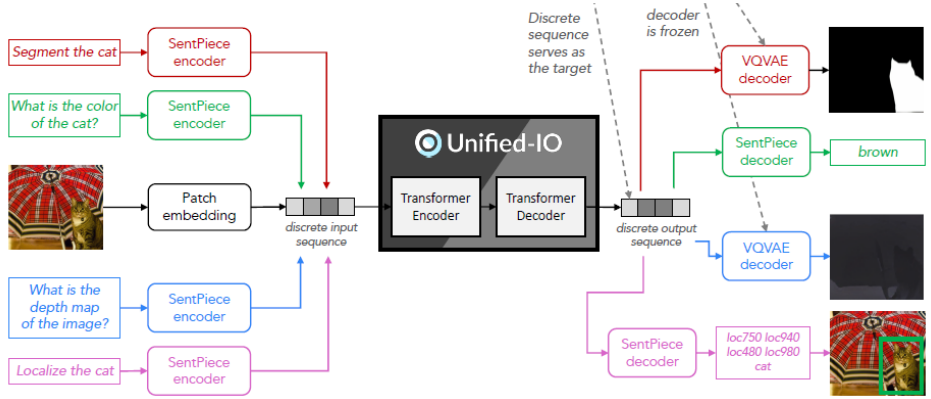
Transformers, Vision and Language (L19)

- Transformers are general data processors, applicable to text, vision, audio, control, and other domains
- Pre-training to generate missing tokens in unsupervised text data learns a general model that can be fine-tuned
  - Same idea is also applicable to other domains
- Transformer architectures are state-of-art for vision and language individually
- Arguably, the biggest benefit of transformers is ability to combine information from multiple domains





Pre-training



CLIP and GPT (L20)

- Deep learning application often involves starting with a pre-trained “foundation” model and fine-tuning it
- With large-scale training and the right formulations, models can perform a range of tasks including those not explicitly trained
- GPT demonstrates that learning to predict the next word produces a flexible zero-shot and few-shot general language task performer
- CLIP shows that learning to match images to text produces a good zero-shot classifier and an excellent image encoder