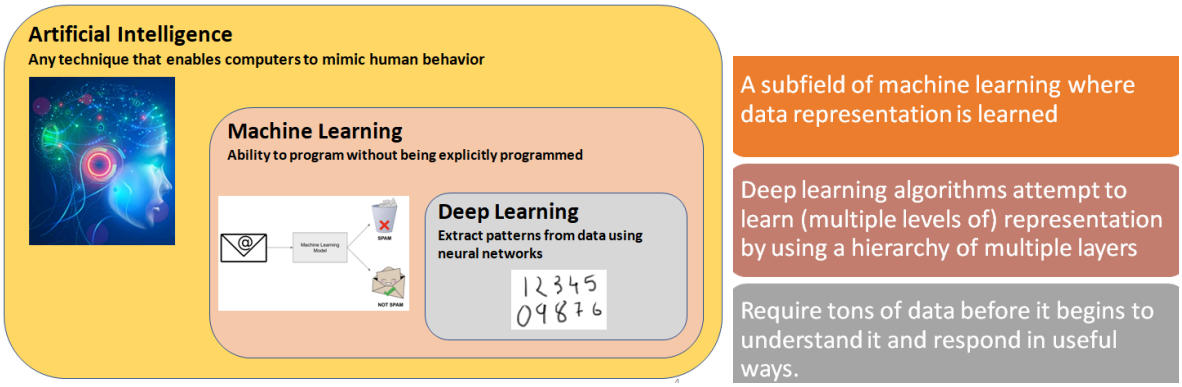
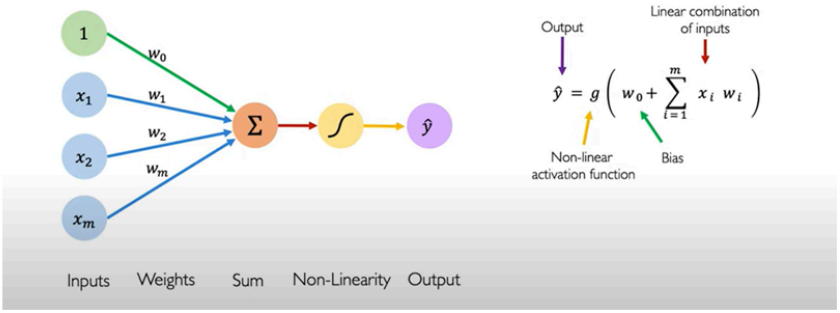
DL
KERNEL, BACKPROPAGATION

What is Deep learning?



•in the past decade only, the amount of data has grown so much so that we can say that we are living in the era of big data, and deep learning requires large amount of data to work efficiently. So it is one of the driving force to the much use of DL now-adays. Secondly, the DL models are parallelizable and thus can benefit from GPUs (that is modern hardware). Such hardware didn't exist in the past when all NN architectures were proposed.
Thirdly, now we have open source toolboxes like tensorflow using which we can build and deploy DL algorithms.

Perceptron (Single layer neural network) is the structural building block of deep learning models



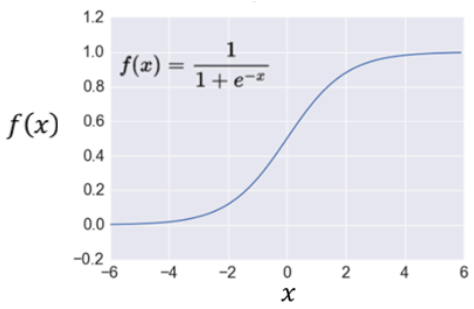
•Non-linear functions approximate complex functions that make neural networks extremely powerful

ONLY RELU DOESN'T HAVE VANISHING GRAD PROBLEM

Activation: Sigmoid

Introduction to Neural Networks

- ***Sigmoid function*** σ: takes a real-valued number and "squashes" it into the range between 0 and 1
  - The output can be interpreted as the firing rate of a biological neuron
    - Not firing = 0; Fully firing = 1
  - When the neuron's activation are 0 or 1, sigmoid neurons saturate
    - Gradients at these regions are almost zero (almost no signal will flow)
  - Sigmoid activations are less common in modern NNs



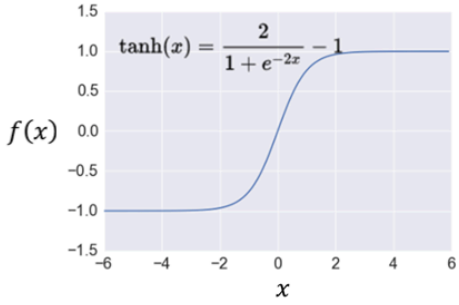
$\mathbb{R}^n \rightarrow [0,1]$

Slide credit: Ismini Lourentzou – Introduction to Deep Learning

Activation: Tanh

Introduction to Neural Networks

- ***Tanh function***: takes a real-valued number and "squashes" it into range between -1 and 1
  - Like sigmoid, tanh neurons saturate
  - Unlike sigmoid, the output is zero-centered
    - It is therefore preferred than sigmoid
  - Tanh is a scaled sigmoid: $\tanh(x) = 2 \cdot \sigma(2x) - 1$



$\mathbb{R}^n \rightarrow [-1,1]$

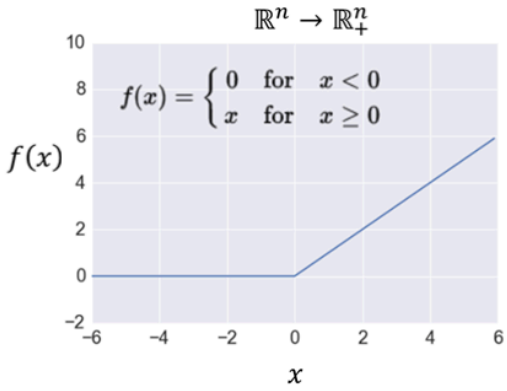
Activation: ReLU

Introduction to Neural Networks

- ***ReLU*** (Rectified Linear Unit): takes a real-valued number and thresholds it at zero

$$f(x) = \max(0, x)$$

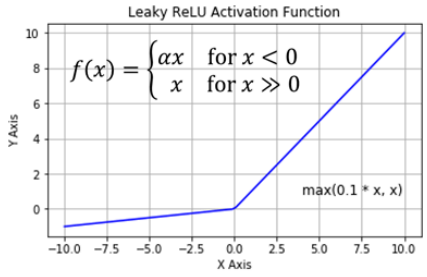
  - Most modern deep NNs use ReLU activations
  - ReLU is fast to compute
    - Compared to sigmoid, tanh
    - Simply threshold a matrix at zero
  - Accelerates the convergence of gradient descent
    - Due to linear, non-saturating form
  - Prevents the gradient vanishing problem



Activation: Leaky ReLU

Introduction to Neural Networks

- The problem of ReLU activations: they can "die"
  - ReLU could cause weights to update in a way that the gradients can become zero and the neuron will not activate again on any data
  - E.g., when a large learning rate is used
- ***Leaky ReLU*** activation function is a variant of ReLU
  - Instead of the function being 0 when $x < 0$, a leaky ReLU has a small negative slope (e.g., $\alpha = 0.01$, or similar)
  - This resolves the dying ReLU problem
  - Most current works still use ReLU
    - With a proper setting of the learning rate, the problem of dying ReLU can be avoided

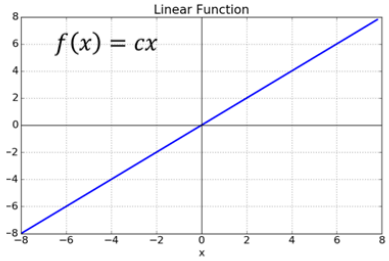


Activation: Linear Function

Introduction to Neural Networks

- ***Linear function*** means that the output signal is proportional to the input signal to the neuron
  - If the value of the constant c is 1, it is also called identity activation function
  - This activation type is used in regression problems
    - E.g., the last layer can have linear activation function, in order to output a real number (and not a class membership)

$\mathbb{R}^n \rightarrow \mathbb{R}^n$



Empirical loss is the total loss incurred by all the inputs. Mean of all loss function of our data set.
While training the network, we want to minimize the empirical loss.
For classification problems, we can use binary cross entropy loss
For regression problems, mean sq error is used.

Training - Minimizing the Loss

The loss function with regard to weights and biases can be defined as

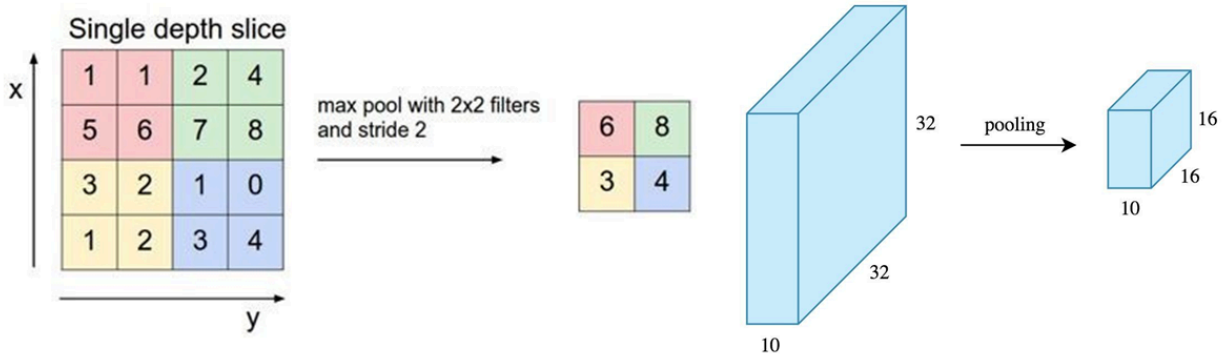
$$L(\mathbf{w}, \mathbf{b}) = \frac{1}{2} \sum_i (\mathbf{Y}(\mathbf{X}, \mathbf{w}, \mathbf{b}) - \mathbf{Y}'(\mathbf{X}, \mathbf{w}, \mathbf{b}))^2$$

The weight update is computed by moving a step to the opposite direction of the cost gradient.

$$\Delta w_i = -\alpha \frac{\partial L}{\partial w_i}$$

Iterate until L stops decreasing.

Max Pooling



Pooling layer is used in CNNs to reduce the spatial dimensions (width and height) of the input feature maps while retaining the most important information. It involves sliding a two-dimensional filter over each channel of a feature map and summarizing the features within the region covered by the filter.

Max Pooling, Sum Pooling

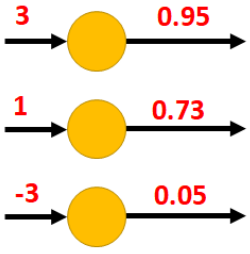
A fully connected layer, also known as a linear layer or dense layer, is a neural network layer that connects every input neuron to every output neuron.

Softmax Layer

Introduction to Neural Networks

- In multi-class classification tasks, the output layer is typically a *softmax layer*
  - I.e., it employs a *softmax activation function*
  - If a layer with a sigmoid activation function is used as the output layer instead, the predictions by the NN may not be easy to interpret
    - Note that an output layer with sigmoid activations can still be used for binary classification

A Layer with Sigmoid Activations



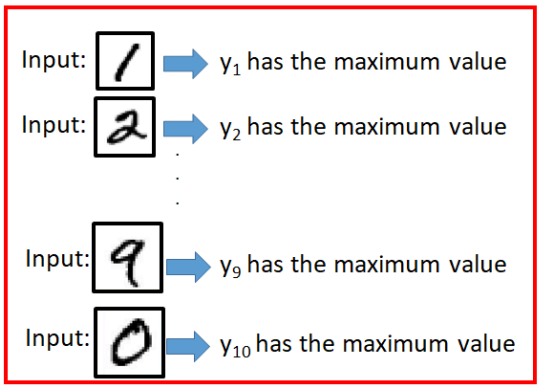
Training an NN

PREPROCESS: Normalize (divide by std dev), Subtract mean (zero centric)

Training NNs

Training Neural Networks

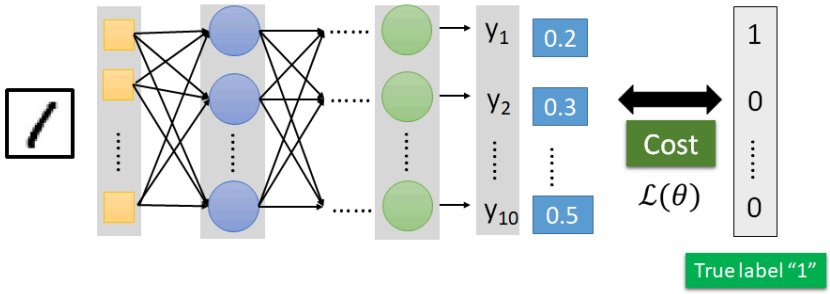
- To train a NN, set the parameters θ such that for a training subset of images, the corresponding elements in the predicted output have maximum values



Training NNs

Training Neural Networks

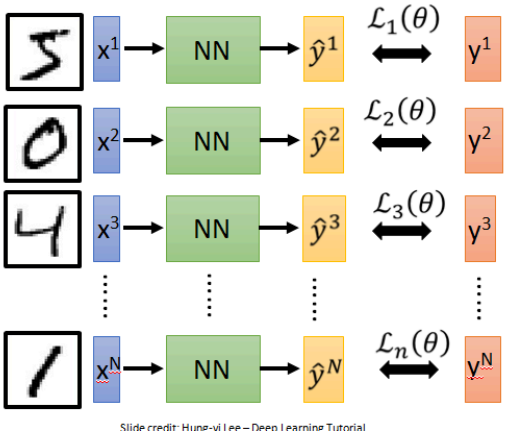
- Define a *loss function*/objective function/cost function $\mathcal{L}(\theta)$ that calculates the difference (error) between the model prediction and the true label
  - E.g., $\mathcal{L}(\theta)$ can be mean-squared error, cross-entropy, etc.



Training NNs

Training Neural Networks

- For a training set of N images, calculate the total loss overall all images: $\mathcal{L}(\theta) = \sum_{n=1}^{N} \mathcal{L}_n(\theta)$
- Find the optimal parameters θ^* that minimize the total loss $\mathcal{L}(\theta)$



Slide credit: Hung-yi Lee – Deep Learning Tutorial

Loss Functions

Training Neural Networks

- ***Classification tasks***

Training examples — Pairs of N inputs x_i and ground-truth class labels y_i

Output Layer — Softmax Activations [maps to a probability distribution]

$$P(y = j \mid \mathbf{x}) = \frac{e^{\mathbf{x}^\top \mathbf{w}_j}}{\sum_{k=1}^{K} e^{\mathbf{x}^\top \mathbf{w}_k}}$$

Loss function — ***Cross-entropy***

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \sum_{k=1}^{K} \left[y_k^{(i)} \log \hat{y}_k^{(i)} + \left(1 - y_k^{(i)}\right) \log\left(1 - \hat{y}_k^{(i)}\right) \right]$$

Ground-truth class labels y_i and model predicted class labels $\hat{y}_i$

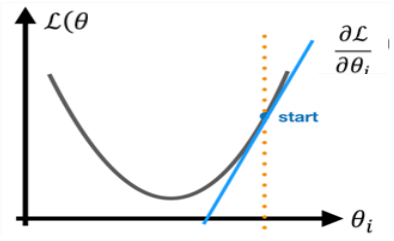
- *Regression tasks*

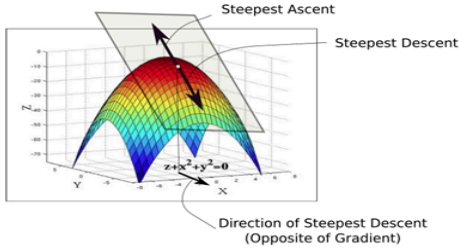
Training examples	Pairs of N inputs x_i and ground-truth output values y_i	
Output Layer	Linear (Identity) or Sigmoid Activation	
Loss function	***Mean Squared Error***	$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$
	Mean Absolute Error	$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left\|y^{(i)} - \hat{y}^{(i)}\right\|$

Training NNs

Training Neural Networks

- Optimizing the loss function $\mathcal{L}(\theta)$
  - Almost all DL models these days are trained with a variant of the ***gradient descent*** (GD) algorithm
  - GD applies iterative refinement of the network parameters θ
  - GD uses the opposite direction of the gradient of the loss with respect to the NN parameters (i.e., $\nabla\mathcal{L}(\theta) = [\partial\mathcal{L}/\partial\theta_i]$) for updating θ
    - The gradient of the loss function $\nabla\mathcal{L}(\theta)$ gives the direction of fastest increase of the loss function $\mathcal{L}(\theta)$ when the parameters θ are changed

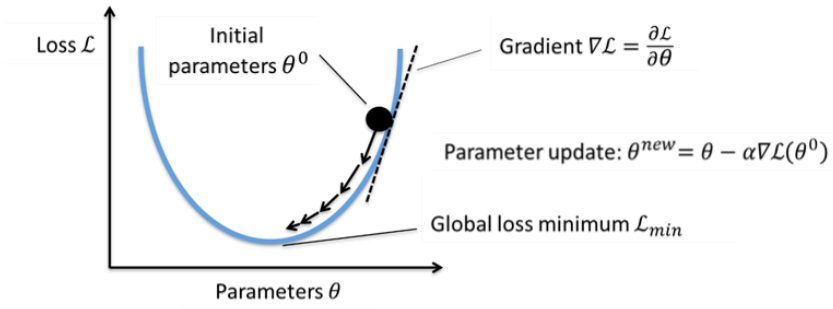




Gradient Descent Algorithm

Training Neural Networks

- Steps in the ***gradient descent algorithm***:
  1. Randomly initialize the model parameters, θ^0
  2. Compute the gradient of the loss function at the initial parameters θ^0: $\nabla\mathcal{L}(\theta^0)$
  3. Update the parameters as: $\theta^{new} = \theta^0 - \alpha\nabla\mathcal{L}(\theta^0)$
     - Where α is the learning rate
  4. Go to step 2 and repeat (until a terminating criterion is reached)



•Gradient descent algorithm stops when a local minimum of the loss surface is reached
•Gradient descent (GD) is an iterative first-order optimization algorithm, used to find a local minimum/maximum of a given function. This method is commonly used in machine learning (ML) and deep learning (DL) to minimize a cost/loss function (e.g. in a linear regression).
§GD does not guarantee reaching a global minimum
However, empirical evidence suggests that GD works well for NNs

Backpropagation

Training Neural Networks

- Modern NNs employ the ***backpropagation*** method for calculating the gradients of the loss function $\nabla\mathcal{L}(\theta) = \partial\mathcal{L}/\partial\theta_i$
  - Backpropagation is short for "backward propagation"
- For training NNs, forward propagation (forward pass) refers to passing the inputs x through the hidden layers to obtain the model outputs (predictions) y
  - The loss $\mathcal{L}(y, \hat{y})$ function is then calculated
  - Backpropagation traverses the network in reverse order, from the outputs y backward toward the inputs x to calculate the gradients of the loss $\nabla\mathcal{L}(\theta)$
  - The chain rule is used for calculating the partial derivatives of the loss function with respect to the parameters θ in the different layers in the network
- Each update of the model parameters θ during training takes one forward and one backward pass (e.g., of a batch of inputs)
- Automatic calculation of the gradients (automatic differentiation) is available in all current deep learning libraries
  - It significantly simplifies the implementation of deep learning algorithms, since it obviates deriving the partial derivatives of the loss function by hand

Feature	Batch Gradient Descent	Stochastic Gradient Descent (SGD)	Mini-Batch Gradient Descent
Definition	Computes gradient using the entire dataset	Computes gradient using one data point	Computes gradient using a small batch of data points
Update Frequency	After processing the entire dataset	After each individual data point	After processing each mini-batch
Convergence Speed	Slower convergence, smoother path	Faster convergence, more noisy path	Balance between convergence speed and noise
Computational Efficiency	High memory usage, efficient in vectorized operations	Low memory usage, less efficient with vectorized operations	Balanced memory usage and computational efficiency
Stability	More stable, less noise in updates	Less stable, high variance in updates	Medium stability, less variance than SGD
Use Cases	Suitable for smaller datasets	Suitable for large-scale, online learning, or real-time applications	Suitable for large datasets and when trade-off between speed and accuracy is needed
Parallelism	Difficult to parallelize	Easy to parallelize	Easier to parallelize than batch gradient descent
Memory Requirements	Requires memory for entire dataset	Requires memory for only one data point	Requires memory for a mini-batch
Risk of Overfitting	Lower risk of overfitting	Higher risk of overfitting due to high variance	Moderate risk of overfitting
Example of Use	Traditional machine learning tasks with small to medium datasets	Online advertising, large-scale recommendation systems	Deep learning training, large-scale machine learning tasks

Types of Gradient Descent (IMP)

Gradient Descent with Momentum

Training Neural Networks

- Parameters update in ***GD with momentum*** at iteration t: $\theta^t = \theta^{t-1} - V^t$
  - Where: $V^t = \beta V^{t-1} + \alpha \nabla \mathcal{L}(\theta^{t-1})$
  - I.e., $\theta^t = \theta^{t-1} - \alpha \nabla \mathcal{L}(\theta^{t-1}) - \beta V^{t-1}$
- Compare to vanilla GD: $\theta^t = \theta^{t-1} - \alpha \nabla \mathcal{L}(\theta^{t-1})$
  - Where θ^{t-1} are the parameters from the previous iteration $t-1$
- The term V^t is called momentum
  - This term accumulates the gradients from the past several steps, i.e.,

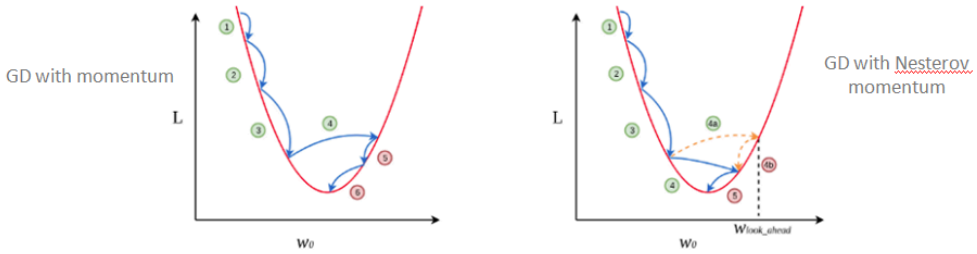
$$\begin{aligned} V^t &= \beta V^{t-1} + \alpha \nabla \mathcal{L}(\theta^{t-1}) \\ &= \beta\left(\beta V^{t-2} + \alpha \nabla \mathcal{L}(\theta^{t-2})\right) + \alpha \nabla \mathcal{L}(\theta^{t-1}) \\ &= \beta^2 V^{t-2} + \beta\alpha \nabla \mathcal{L}(\theta^{t-2}) + \alpha \nabla \mathcal{L}(\theta^{t-1}) \\ &= \beta^3 V^{t-3} + \beta^2 \alpha \nabla \mathcal{L}(\theta^{t-3}) + \beta\alpha \nabla \mathcal{L}(\theta^{t-2}) + \alpha \nabla \mathcal{L}(\theta^{t-1}) \end{aligned}$$

  - This term is analogous to a momentum of a heavy ball rolling down the hill
- The parameter β is referred to as a coefficient of momentum
  - A typical value of the parameter β is 0.9
- This method updates the parameters θ in the direction of the weighted average of the past gradients

Nesterov Accelerated Momentum

Training Neural Networks

- ***Gradient descent with Nesterov accelerated momentum***
  - Parameter update: $\theta^t = \theta^{t-1} - V^t$
    - Where: $V^t = \beta V^{t-1} + \alpha \nabla \mathcal{L}(\theta^{t-1} + \beta V^{t-1})$
  - The term $\theta^{t-1} + \beta V^{t-1}$ allows to predict the position of the parameters in the next step (i.e., $\theta^t \approx \theta^{t-1} + \beta V^{t-1}$)
  - The gradient is calculated with respect to the approximate future position of the parameters in the next iteration, θ^t, calculated at iteration $t-1$



Adam

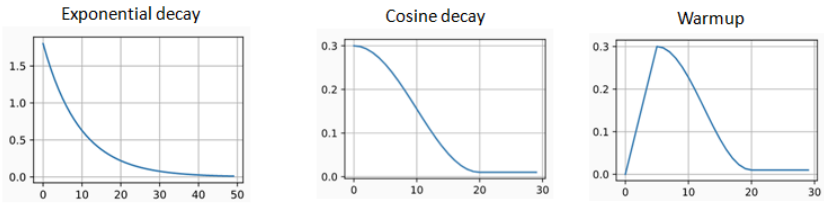
Training Neural Networks

- ***Adaptive Moment Estimation (Adam)***
  - Adam combines insights from the momentum optimizers that accumulate the values of past gradients, and it also introduces new terms based on the second moment of the gradient
    - Similar to GD with momentum, Adam computes a **weighted average of past gradients** (first moment of the gradient), i.e., $V^t = \beta_1 V^{t-1} + (1-\beta_1)\nabla \mathcal{L}(\theta^{t-1})$
    - Adam also computes a **weighted average of past squared gradients** (second moment of the gradient), , i.e., $U^t = \beta_2 U^{t-1} + (1-\beta_2)\left(\nabla \mathcal{L}(\theta^{t-1})\right)^2$
  - The parameter update is: $\theta^t = \theta^{t-1} - \alpha \frac{\hat{V}^t}{\sqrt{\hat{U}^t} + \epsilon}$
    - Where: $\hat{V}^t = \frac{V^t}{1-\beta_1}$ and $\hat{U}^t = \frac{U^t}{1-\beta_2}$
    - The proposed default values are $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$
- Other commonly used optimization methods include:
  - Adagrad, Adadelta, RMSprop, Nadam, etc.
  - Most commonly used optimizers nowadays are Adam and SGD with momentum

Learning Rate Scheduling

Training Neural Networks

- ***Learning rate scheduling*** is applied to change the values of the learning rate during the training
  - ***Annealing*** is reducing the learning rate over time (a.k.a. learning rate decay)
    - Approach 1: reduce the learning rate by some factor every few epochs
      - Typical values: reduce the learning rate by a half every 5 epochs, or divide by 10 every 20 epochs
    - Approach 2: exponential or cosine decay gradually reduce the learning rate over time
    - Approach 3: reduce the learning rate by a constant (e.g., by half) whenever the validation loss stops improving
      - In TensorFlow: tf.keras.callbacks.ReduceLROnPleateau()
        - Monitor: validation loss, factor: 0.1 (i.e., divide by 10), patience: 10 (how many epochs to wait before applying it), Minimum learning rate: 1e-6 (when to stop)
  - ***Warmup*** is gradually increasing the learning rate initially, and afterward let it cool down until the end of the training



Vanishing Gradient v/s Exploding Gradient (IMP)

Aspect	Vanishing Gradient	Exploding Gradient
Problem Definition	Gradients become extremely small, nearing zero.	Gradients become extremely large, approaching infinity.
Effect on Training	Slow or stalled learning, weights barely change.	Unstable training, weights grow uncontrollably.
Cause	Activation functions (like sigmoid or tanh) squash gradients, especially for deep networks.	Large weights or unnormalized gradients lead to runaway updates.
Common in	Deep neural networks with many layers.	Deep neural networks, especially with poor weight initialization or large learning rates.
Impact on Learning	Network fails to learn meaningful representations for layers deeper in the network.	Learning becomes unstable, leading to diverging values.
Mathematical Representation	Gradients tend to zero as backpropagation is performed over many layers.	Gradients grow exponentially through backpropagation.
Mitigation Strategies	Use ReLU (or variants), gradient clipping, batch normalization, skip connections.	Gradient clipping, careful weight initialization (e.g., Xavier/He initialization), lower learning rates.
Typical Symptoms	Slow convergence or no learning after some point.	Large gradients causing NaNs or weights becoming too large.

	L1 regularization	L2 regularization
How it works	Adds the absolute value of coefficients to the loss function	Adds the squared value of coefficients to the loss function
What it does	Promotes sparsity and feature selection	Encourages smaller, more evenly distributed weights
When to use	When a model needs fewer features	When all features need consideration

L1 Regularization

$$\text{Cost} = \sum_{i=0}^{N}(y_i - \sum_{j=0}^{M} x_{ij}W_j)^2 + \lambda \sum_{j=0}^{M} |W_j|$$

L2 Regularization

$$\text{Cost} = \sum_{i=0}^{N}(y_i - \sum_{j=0}^{M} x_{ij}W_j)^2 + \lambda \sum_{j=0}^{M} W_j^2$$

Loss function

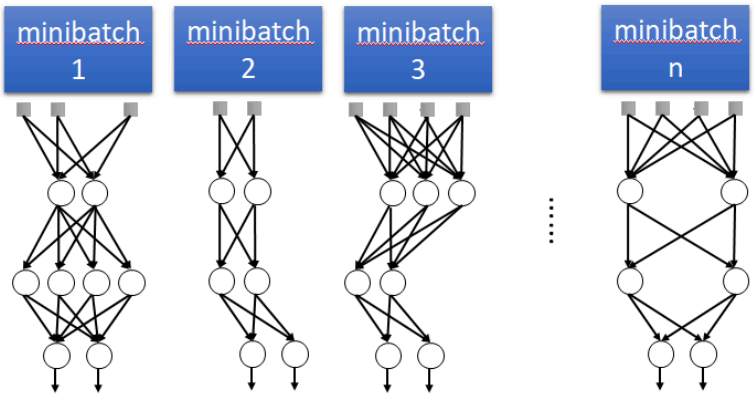
Regularization Term

L1: elastic net

Regularization: Dropout

Regularization

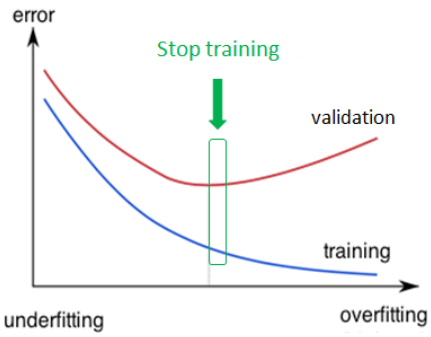
- Dropout is a kind of ensemble learning
  - Using one mini-batch to train one network with a slightly different architecture



Regularization: Early Stopping

Regularization

- ***Early-stopping***
  - During model training, use a validation set
    - E.g., validation/train ratio of about 25% to 75%
  - Stop when the validation accuracy (or loss) has not improved after n epochs
    - The parameter n is called patience



Hyper-parameter Tuning

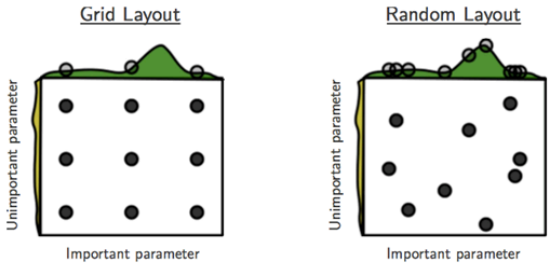
Hyper-parameter Tuning

- Training NNs can involve setting many ***hyper-parameters***
- The most common hyper-parameters include:
  - Number of layers, and number of neurons per layer
  - Initial learning rate
  - Learning rate decay schedule (e.g., decay constant)
  - Optimizer type
- Other hyper-parameters may include:
  - Regularization parameters (ℓ_2 penalty, dropout rate)
  - Batch size
  - Activation functions
  - Loss function
- Hyper-parameter tuning can be time-consuming for larger NNs

Hyper-parameter Tuning

Hyper-parameter Tuning

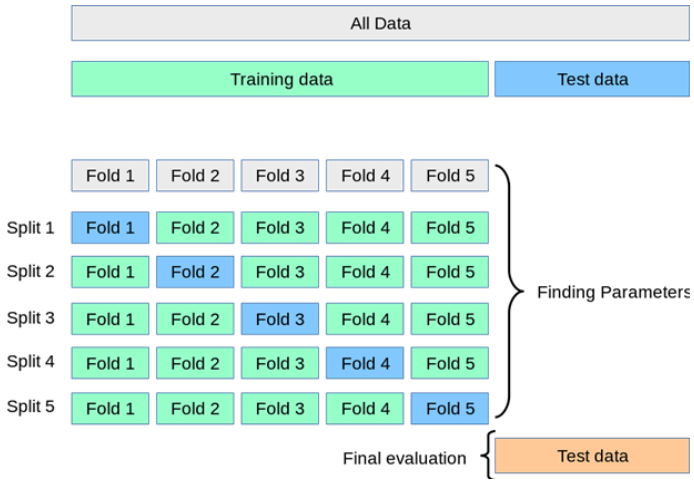
- Grid search
  - Check all values in a range with a step value
- Random search
  - Randomly sample values for the parameter
  - Often preferred to grid search
- Bayesian hyper-parameter optimization
  - Is an active area of research



k-Fold Cross-Validation

k-Fold Cross-Validation

- Using ***k-fold cross-validation*** for hyper-parameter tuning is common when the size of the training data is small
  - It also leads to a better and less noisy estimate of the model performance by averaging the results across several folds
- E.g., 5-fold cross-validation (see the figure on the next slide)
  1. Split the train data into 5 equal folds
  2. First use folds 2-5 for training and fold 1 for validation
  3. Repeat by using fold 2 for validation, then fold 3, fold 4, and fold 5
  4. Average the results over the 5 runs (for reporting purposes)
  5. Once the best hyper-parameters are determined, evaluate the model on the test data



Ensemble Learning

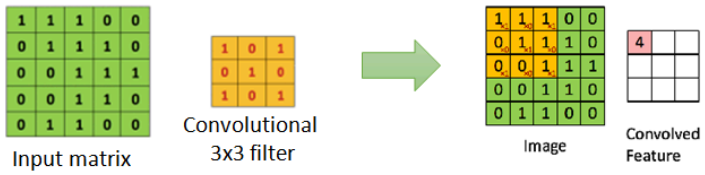
Ensemble Learning

- ***Ensemble learning*** is training multiple classifiers separately and combining their predictions
  - Ensemble learning often outperforms individual classifiers
  - Better results obtained with higher model variety in the ensemble
  - ***Bagging*** (bootstrap aggregating)
    - Randomly draw subsets from the training set (i.e., bootstrap samples)
    - Train separate classifiers on each subset of the training set
    - Perform classification based on the average vote of all classifiers
  - ***Boosting***
    - Train a classifier, and apply weights on the training set (apply higher weights on misclassified examples, focus on "hard examples")
    - Train new classifier, reweight training set according to prediction error
    - Repeat
    - Perform classification based on weighted vote of the classifiers

Convolutional Neural Networks (CNNs)

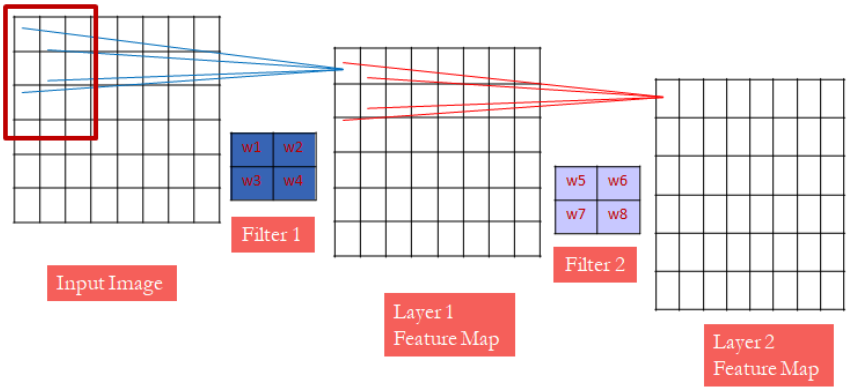
Convolutional Neural Networks

- ***Convolutional neural networks*** (CNNs) were primarily designed for image data
- CNNs use a convolutional operator for extracting data features
  - Allows parameter sharing
  - Efficient to train
  - Have less parameters than NNs with fully-connected layers
- CNNs are robust to spatial translations of objects in images
- A convolutional filter slides (i.e., convolves) across the image



In CNNs, hidden units in a layer are only connected to a small region of the layer before it (called local receptive field)

- The depth of each feature map corresponds to the number of convolutional filters used at each layer



Feature extraction architecture

- After 2 convolutional layers, a max-pooling layer reduces the size of the feature maps (typically by 2)
- A fully convolutional and a softmax layers are added last to perform classification

Residual Networks (ResNets) are a type of deep neural network architecture designed to address the problem of vanishing gradients in very deep networks. They introduce **"identity" skip connections**, which allow the input of a layer to bypass one or more layers and be added directly to the output.

Skip Connections (Identity Mapping):

Instead of learning the full transformation of the input at each layer, ResNets allow layers to learn residual functions. That is, instead of computing H(x)H(x)H(x), they learn the residual F(x)F(x)F(x) where:

H(x)=F(x)+xH(x) = F(x) + xH(x)=F(x)+x

This helps the gradient flow backward more easily, mitigating vanishing gradients.

Deep Networks Become Trainable:

Thanks to skip connections, ResNets enable training of very deep networks—**over 1,000 layers**—which was previously difficult due to vanishing gradients.

Multiple Variants:

Different ResNet architectures exist, primarily varying in depth: **ResNet-18, 34, 50, 101, 152, and 200** layers.

- The difference is in the number of layers and the use of **bottleneck blocks** (which reduce computational cost in deeper versions like ResNet-50 and beyond).

Recurrent Neural Networks (RNNs)

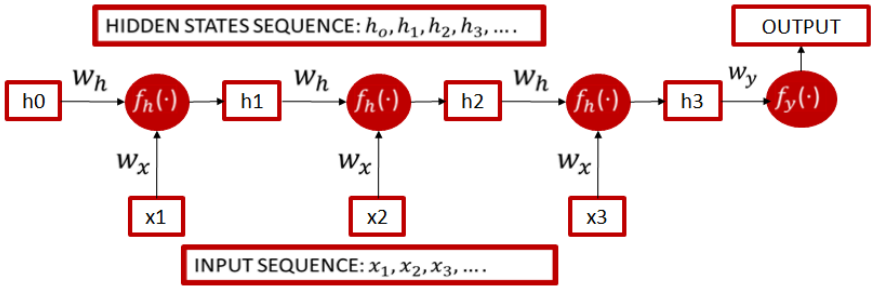
Recurrent Neural Networks

- ***Recurrent NNs*** are used for modeling sequential data and data with varying length of inputs and outputs
  - Videos, text, speech, DNA sequences, human skeletal data
- RNNs introduce recurrent connections between the neurons
  - This allows processing sequential data one element at a time by selectively passing information across a sequence
  - Memory of the previous inputs is stored in the model's internal state and affect the model predictions
  - Can capture correlations in sequential data
- RNNs use backpropagation-through-time for training
- RNNs are more sensitive to the vanishing gradient problem than CNNs

Recurrent Neural Networks (RNNs)

Recurrent Neural Networks

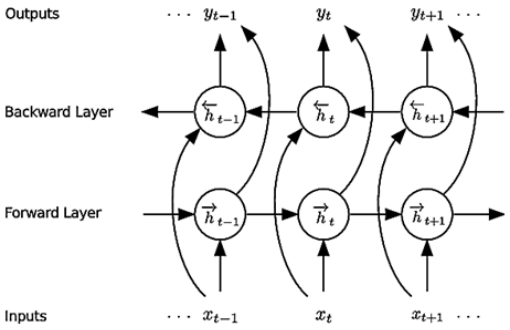
- RNN use same set of weights w_h and w_x across all time steps
  - A sequence of hidden states $\{h_o, h_o h_2, h_3, \dots\}$ is learned, which represents the memory of the network
  - The hidden state at step t, $h(t)$, is calculated based on the previous hidden state $h(t-1)$ and the input at the current step $x(t)$, i.e., $h(t) = f_h\big(w_h * h(t-1) + w_x * x(t)\big)$
  - The function $f_h(\cdot)$ is a nonlinear activation function, e.g., ReLU or tanh
- RNN shown rolled over time



Bidirectional RNNs

Recurrent Neural Networks

- ***Bidirectional RNNs*** incorporate both forward and backward passes through sequential data
  - The output may not only depend on the previous elements in the sequence, but also on future elements in the sequence
  - It resembles two RNNs stacked on top of each other



$$\vec{h}_t = \sigma(\vec{W}^{(hh)}\vec{h}_{t-1} + \vec{W}^{(hx)}x_t)$$

$$\overleftarrow{h}_t = \sigma(\overleftarrow{W}^{(hh)}\overleftarrow{h}_{t+1} + \overleftarrow{W}^{(hx)}x_t)$$

$$y_t = f([\vec{h}_t; \overleftarrow{h}_t])$$

Outputs both past and future elements

LSTM Networks

Recurrent Neural Networks

- ***Long Short-Term Memory (LSTM)*** networks are a variant of RNNs
- LSTM mitigates the vanishing/exploding gradient problem
  - Solution: a Memory Cell, updated at each step in the sequence
- Three gates control the flow of information to and from the Memory Cell
  - Input Gate: protects the current step from irrelevant inputs
  - Output Gate: prevents current step from passing irrelevant information to later steps
  - Forget Gate: limits information passed from one cell to the next
- Most modern RNN models use either LSTM units or other more advanced types of recurrent units (e.g., GRU units)

WHY CNN?

Disadvantage of ann:

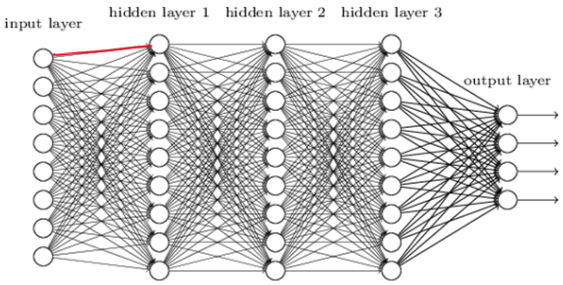
•The number of trainable parameters becomes extremely large

•Require more computational and memory requirements

Problems with Fully connected NN

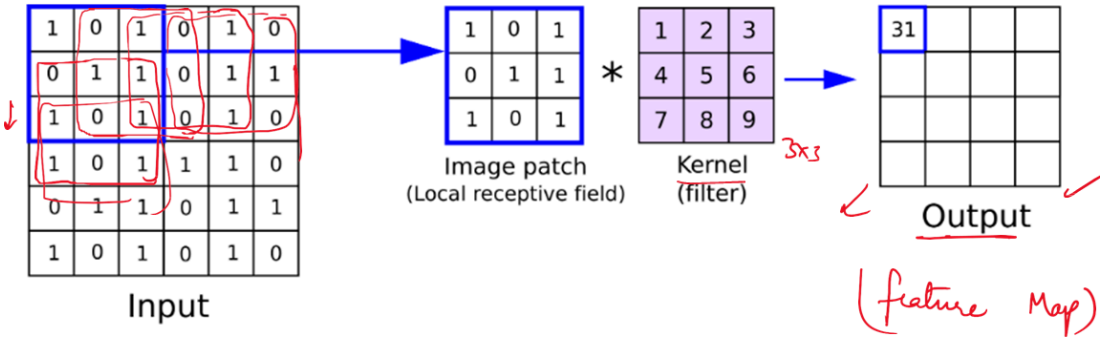
- Fully connected (FC) network: Dense connections
- Too many parameters/weights to learn
- Prone to overfitting
- Vanishing gradient problem

Fully connected NN is good to learn a small model



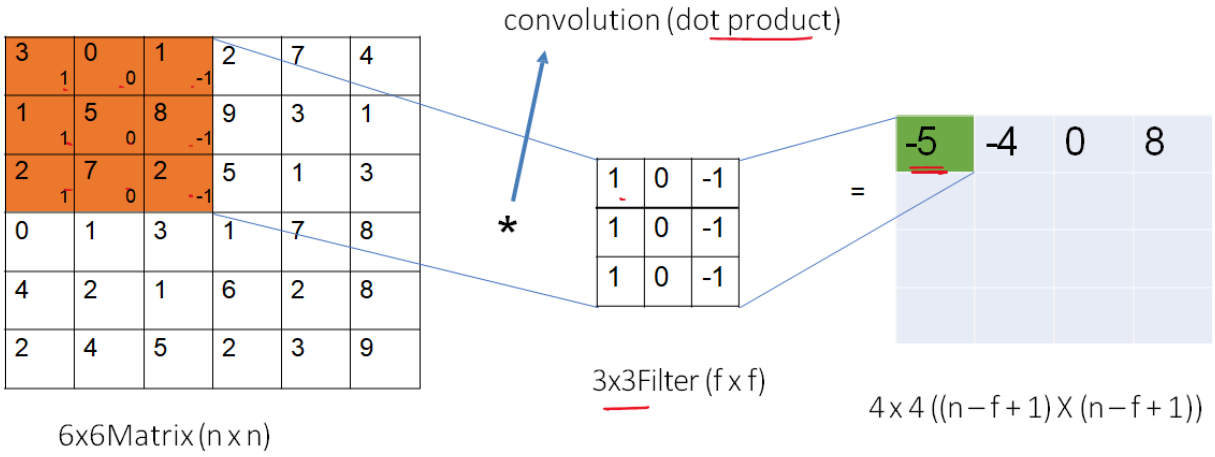
Convolution means applying filters to the input to create a feature map

Convolution Operation



The output is called a **feature map**.
Once a feature map is created, we can pass each value in the feature map through a nonlinearity, such as a ReLU, much like we do for the outputs of a fully connected layer.

Convolution Operation: Example



This property of CNN is termed as sparse connectivity i.e., most of the connections have 0 weights.

2nd property:

Weight Sharing in CNN

The filter is applied in a "sliding window" across the entire layer's input.

The "weight sharing" is using fixed weights for this filter across the entire input.

Input $X = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}$

filter $F = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}$

$$\beta = [w_{11}, w_{12}, w_{21}, w_{22}]$$

$$F * X = \begin{bmatrix} \beta \cdot [x_{11}, x_{12}, x_{21}, x_{22}] & \beta \cdot [x_{12}, x_{13}, x_{22}, x_{23}] \\ \beta \cdot [x_{21}, x_{22}, x_{31}, x_{32}] & \beta \cdot [x_{22}, x_{23}, x_{32}, x_{33}] \end{bmatrix}$$

Two Properties:

CNNs have sparse connectivity: For each layer, each output value depends on a small num of inputs, instead of all inputs

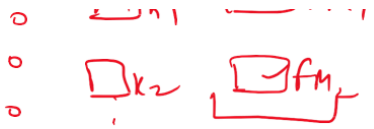
CNNs use weight sharing: Single filter convolved over the entire image

Why 3x3 Convolution filters?

Usually, 3x3 or 5x5 size is taken for convolution filters.

1. Most of the features in an image are usually local. Therefore, it makes sense to take a few local pixels at once and apply convolutions.
2. Most of the features may be found in more than one place in an image. This means that it makes sense to use a single kernel all over the image, hoping to extract that feature in different parts of the image.
3. The number of parameters grows quadratically with kernel size. This makes big convolution kernels not cost-efficient enough.
4. By limiting the number of parameters, we are limiting the number of unrelated features possible.

Feature learning



Would the model be able to learn the features using a single filter and perform well on training and test set?

No, we use multiple filters to capture multiple representations of the input (say 1 filter capturing edges, 1 filter for the purpose of sharpening the input and so on). Let's say we use 100 filters of size 2X2, so the total number of parameters would be 100*4 = 400 which is not bad as we would have many such intermediate layers and each layer would have its own set of weights.

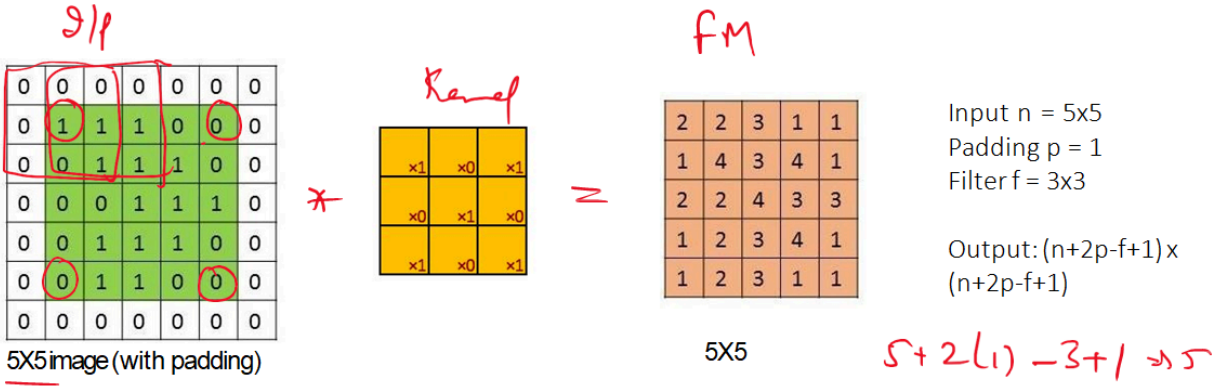
Disadvantages of Convolution

1. Every time we apply a convolutional operation, the size of the image shrinks
2. Pixels present in the corner of the image are used only a few times during convolution as compared to the central pixels. Hence, we do not focus too much on the corners.

To overcome these issues, we use padding

Padding: Zeros are added to outside of the input

- Padding helps in extracting information lying on the corners of a image.
- In turn, it preserves the original dimensions of the input.
- Number of zero layers depend upon the size of the kernel



Stride

Stride specifies the num of steps we take in horizontal and vertical directions separately by sliding the kernel.
E.g., Here, stride s = 3

3	0	1	2	7	4
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
4	2	1	6	2	8
2	4	5	2	3	9

6X6 Matrix

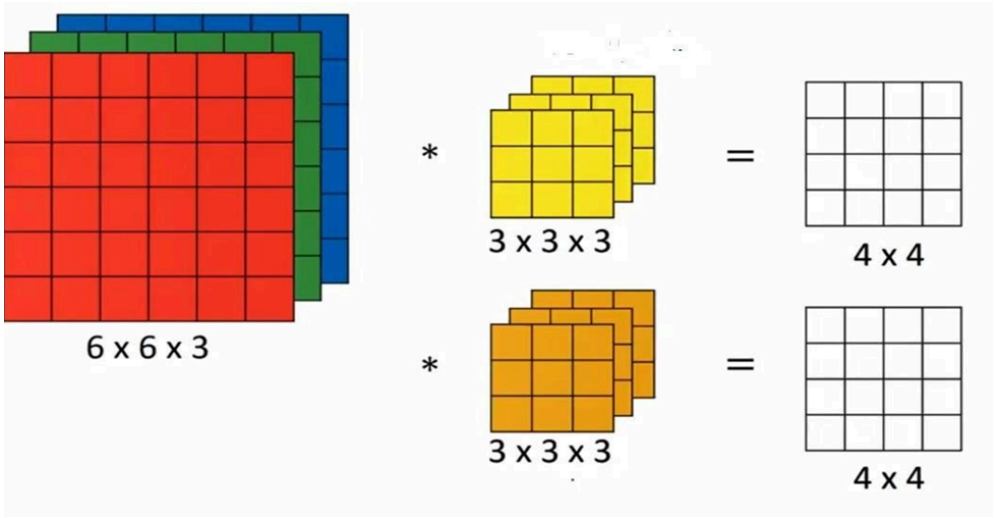
1	0	-1
1	0	-1
1	0	-1

3X3 Filter

2X2

Output: [(n+2p-f)/s+1] X [(n+2p-f)/s+1]

Channels



Generalized dimensions can be given as:

- **Input:** n X n X n_c
- **Filter:** f X f X n_c
- **Padding:** p
- **Stride:** s
- **Output:** [(n+2p-f)/s+1] X [(n+2p-f)/s+1] X n_c'

Here, n_c is the number of channels in the input and filter, while n_c' is the number of filters.

After convolving the filter over the entire image, we add a bias term to those outputs and finally apply an activation function to generate activations.
By default, Relu activation function is used after convolution operation.

One layer of Convolutional network

In CNN, the number of parameters is independent of the size of the image.
It essentially depends on the filter size

Calculating num of parameters in one convolutional layer
Suppose there are 10 filters, each of shape 3 X 3 X 3.

Number of parameters for each filter = 3*3*3 = 27
There will be a bias term for each filter, so total parameters per filter = 28
As there are 10 filters, the total parameters for that layer = 28*10 = 280
No matter how big the image is, the parameters only depend on the filter size.

In a CNN, there are basically three types of layers:

1.Convolution layer
2.Pooling layer
3.Fully connected layer

Pooling

The idea here is that one feature map (output after convolving input with the filter) captures the representation for the entire image (and is typical of the same size as the input if appropriate padding is applied) and to shrink the size of the feature map, we look at a particular neighborhood and whatever be the most active part (highest number) in that neighborhood we keep that.

Reduces the dimensionality of feature map

It speeds up computation

Flattening

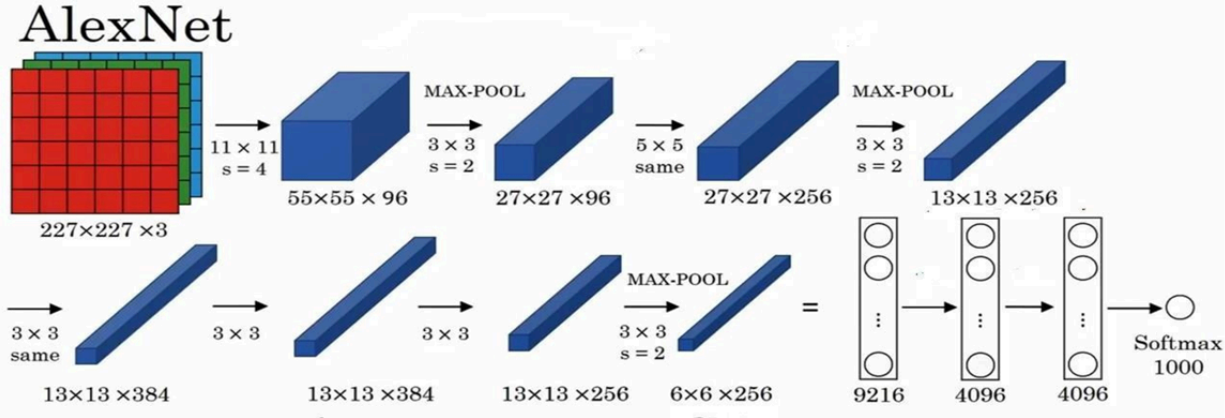
- The process of converting a 2-d array into a vector is called Flattening.
- After a series of convolution and pooling operations on the feature representation of the image, we then flatten the output of the final pooling layers into a single long continuous linear array or a vector.
- Flatten output is fed as input to the fully connected neural network having varying numbers of hidden layers to learn the non-linear complexities present with the feature representation.

Fully connected layer

- The input to the fully connected (FC) layer in the CNN represents the feature vector for the input.
- The aim of the Fully connected layer is to use the high-level feature of the input image produced by convolutional and pooling layers for classifying the input image into various classes based on the training dataset.
- Softmax activation function is used at the output layer.

Hyperparameters in CNN

- Num of filters
- Size of filters
- Stride
- Padding



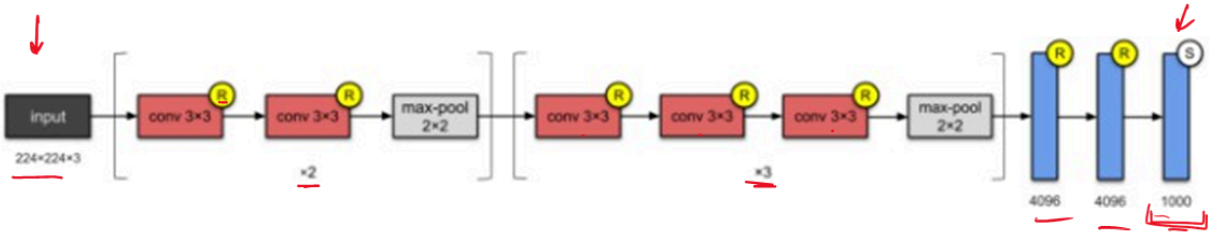
Conv layer 1: 96 filters of size 11x11 with stride = 4 is used, feature map obtained is 55x55x96 ((input-filter)/stride + 1), Activation used -> Relu

MaxPooling layer 1: 3x3, stride = 2

Conv layer 2: 256 filters of size 5x5 used

VGG16

- Proposed in 2014 by Visual Geometry Group (VGG)
- 13 conv layers, 3 FC layers -> total 16 layers
- 138 million parameters

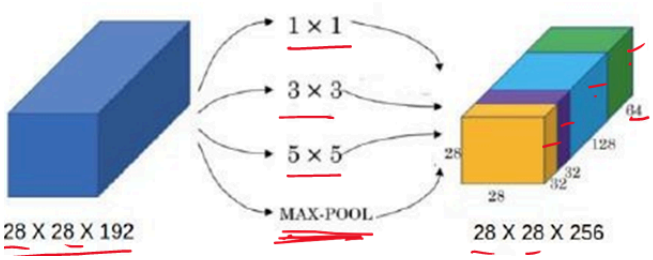


For parameter nums, see ppt 5

Motivation behind Inception network

While designing a convolutional neural network, we have to decide the filter size.
Should it be a 1 X 1 filter, or a 3 X 3 filter, or a 5X 5?
And what should come first: conv or pooling layer?
Inception does all of that for us! Let's see how it works.

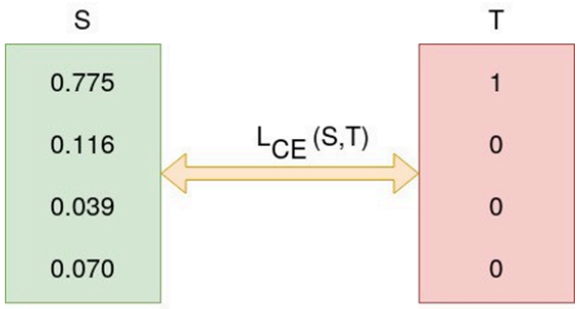
Suppose we have a 28 X 28 X 192 input volume. Instead of choosing what filter size to use, or whether to use convolution layer or pooling layer, inception uses all of them and stacks all the outputs:



Use 64 1x1 conv -> output 28x28x64
Use 128 3x3 conv -> output 28x28x128
Use 32 5x5 conv -> output 28x28x32
Use max pooling with padding to keep the dimensions same (that is 28x28), so that all can be stacked up -> 28x28x32
Final output is 28x28x256

Loss function

Binary Cross Entropy

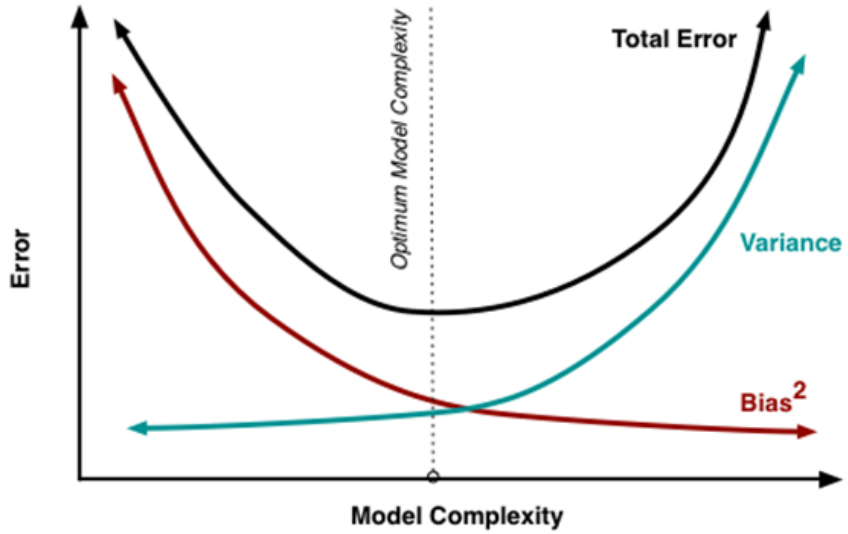


$$
\begin{aligned}
L_{CE} &= -\sum_{i=1} T_i \log(S_i) \\
&= -\left[1 \log_2(0.775) + 0 \log_2(0.126) + 0 \log_2(0.039) + 0 \log_2(0.070)\right] \\
&= -\log_2(0.775) \\
&= 0.3677
\end{aligned}
$$

For softmax, choose cross entropy

Bias vs Variance

- **Bias** is the simplifying assumptions made by the model to make the target function easier to approximate.
- **Variance** is the amount that the estimate of the target function will change, given different training data.



How to try and fix Overfitting

- Reducing model complexity is the best course of action if the model is overfitting.
- Regularization can be used to eliminate higher-order terms from our model.
- Utilizing additional training examples can help the model be trained to generalise better, which is another technique to model fitting.

How to try and avoid underfitting

- Obtain more data.
- Increase the number of epochs.
- Increase model complexity by adding more parameters.
- Replacement of the model is also a good idea.

Region-based CNN (RCNN)

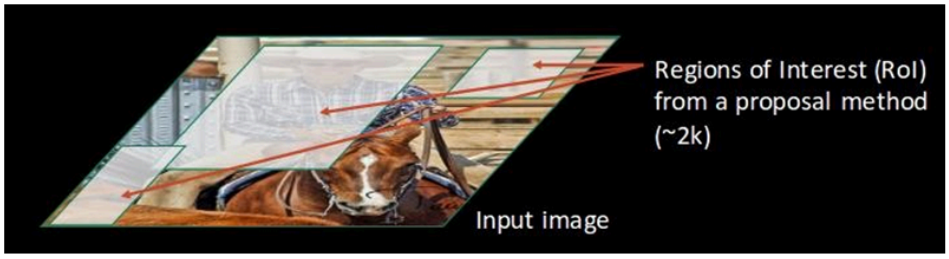
- **RCNN proposes a bunch of boxes in the image and checks if any of these boxes contain any object.**

- RCNN uses **Selective search** for region proposals
  - Takes an image as input
  - Then, it generates initial sub-segmentations so that we have multiple regions from the image
  - It then combines the similar regions to form a larger region (based on color similarity, texture similarity, size similarity, and shape compatibility)
  - Finally, these regions then produce the final object locations (Region of Interest).

Steps of RCNN

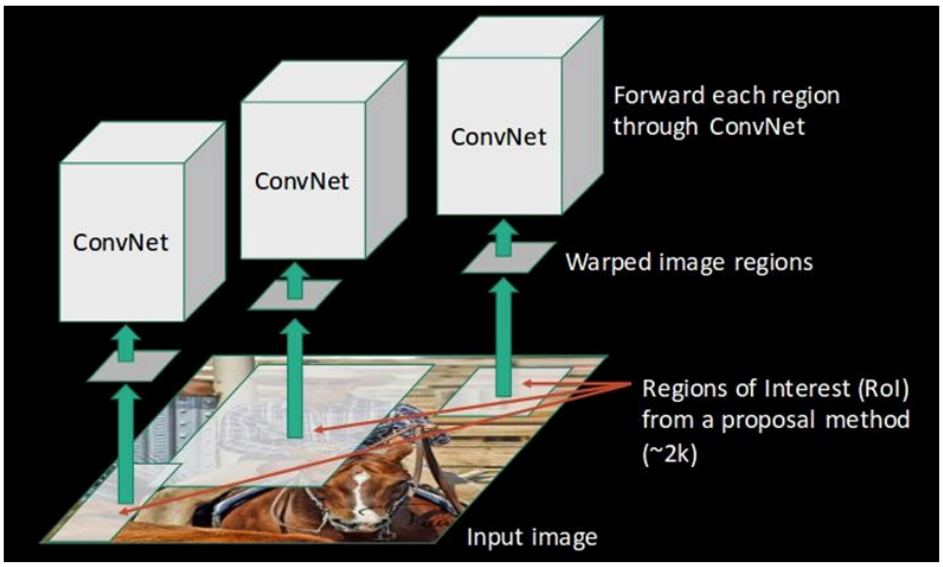
1. First image is taken as input:

2. Then, Regions of Interest (ROI) are proposed using selective search



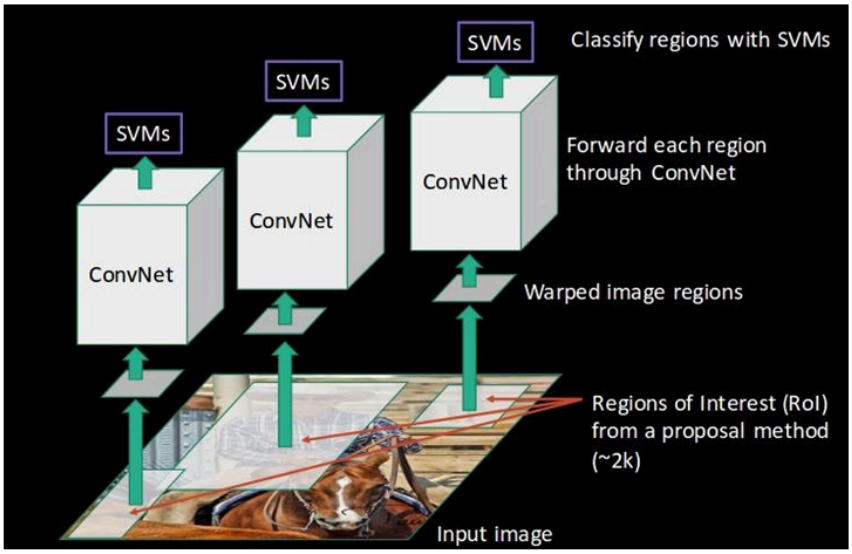
Steps of RCNN

3. All these regions are then reshaped as per the input of the CNN, and each region is passed to the ConvNet



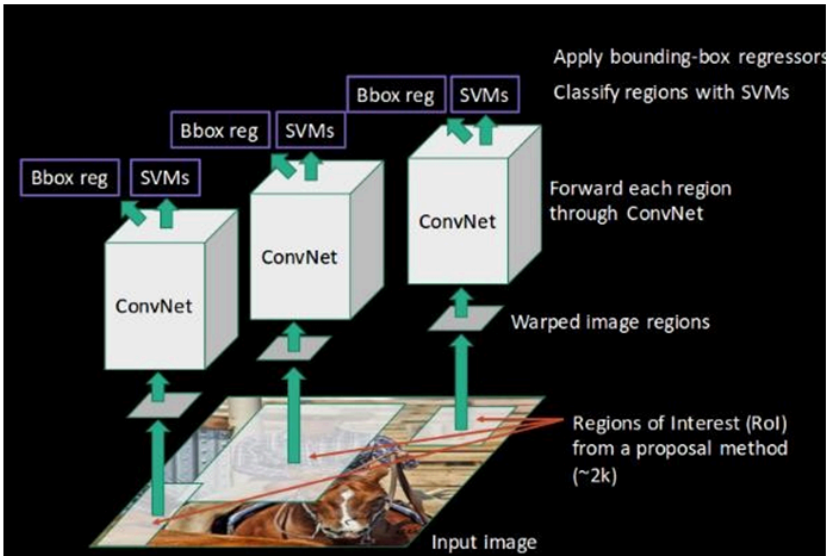
Steps of RCNN

4. CNN then extracts features for each region and SVMs are used to divide these regions into different classes



Steps of RCNN

5. Finally, a bounding box regression is used to predict the tighter bounding boxes for each identified region:



Problems of RCNN:

Training an RCNN model is expensive due to:

- Extracting 2,000 regions for each image based on selective search.
- Extracting features using CNN for every image region. Suppose we have N images, then the number of CNN features will be N*2,000
- The entire process of object detection using RCNN has three models:
  - CNN for feature extraction
  - Linear SVM classifier for identifying objects
  - Regression model for tightening the bounding boxes.

Fast Region-based CNN

- **Objective: Speed up computation. Instead of running a CNN 2,000 times per image, we can run it just once per image and get all the regions of interest.**

- In Fast RCNN, we feed the input image to the CNN, which in turn generates the convolutional feature maps. Using these maps, the regions of proposals are extracted. We then use a RoI pooling layer to reshape all the proposed regions into a fixed size, so that it can be fed into a fully connected network.

Fast Region-based CNN

Steps:

- Take an image as an input
- This image is passed to a ConvNet which in turns generates the Regions of Interest.
- A RoI pooling layer is applied on all of these regions to reshape them as per the input of the ConvNet. Then, each region is passed on to a fully connected network.
- A softmax layer is used on top of the FC network to output classes.
- Also, a linear regression layer is also used parallely to output bounding box coordinates for predicted classes.

Faster Region-based CNN

Faster RCNN is the modified version of Fast RCNN.

Fast RCNN uses selective search for generating Regions of Interest, while Faster RCNN uses "Region Proposal Network" (RPN).

RPN takes image feature maps as an input and generates a set of object proposals, each with an objectness score as output.

Faster Region-based CNN

STEPS:

1. Take an image as input and pass it to the ConvNet which returns the feature maps.
2. Region proposal network is applied on these feature maps. This returns the object proposals along with their objectness score.
3. A RoI pooling layer is applied on these proposals to bring them down to the same size.
4. Finally, the proposals are passed to a fully connected layer which has a softmax layer and a linear regression layer at its top, to classify and output the bounding boxes for objects.

RPN

- Generate anchor boxes.
- Classify each anchor box whether it is foreground or background.
- Learn the shape offsets for anchor boxes to fit them for objects.

Anchor point: Every point in the feature map.
We need to generate anchor boxes for every anchor point.
We generate candidate boxes using two parameters — scales and aspect ratios.

	R-CNN	Fast R-CNN	Faster R-CNN
Test time per image (with proposals)	50 seconds	2 seconds	**0.2 seconds**
(Speedup)	1x	25x	**250x**

Mini-Batch Normalization (BatchNorm) – Explained

Mini-batch normalization (BatchNorm) is a technique used in deep learning to stabilize and accelerate training by **normalizing activations** within a mini-batch. It helps combat **internal covariate shift**, making neural networks train faster and generalize better.

✅ **Speeds up convergence** by reducing dependence on weight initialization.
✅ **Improves stability** by keeping activations in a controlled range.
✅ **Reduces internal covariate shift** (drastic changes in activation distributions).
✅ **Acts as a form of regularization**, reducing the need for dropout in some cases.

Step 1: Compute Mini-Batch Mean & Variance

For a given activation x in a mini-batch of size m:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \quad \text{(Mean)}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \quad \text{(Variance)}$$

Step 2: Normalize Each Activation

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

where ϵ is a small constant to avoid division by zero.

Step 3: Scale and Shift (Learnable Parameters)

Instead of forcing zero mean and unit variance, BatchNorm allows the network to learn optimal scaling and shifting:

$$y_i = \gamma \hat{x}_i + \beta$$

where:

- γ (scale) and β (shift) are learnable parameters.

Layer Normalization (LayerNorm) → Normalizes across feature dimensions (useful for NLP & transformers).
Instance Normalization (InstanceNorm) → Normalizes each image separately (used in style transfer).
Group Normalization (GroupNorm) → Splits channels into groups for normalization (useful for small batch sizes).