ECE521 Lecture10

Deep Learning

UNIVERSITY OF
TORONTO

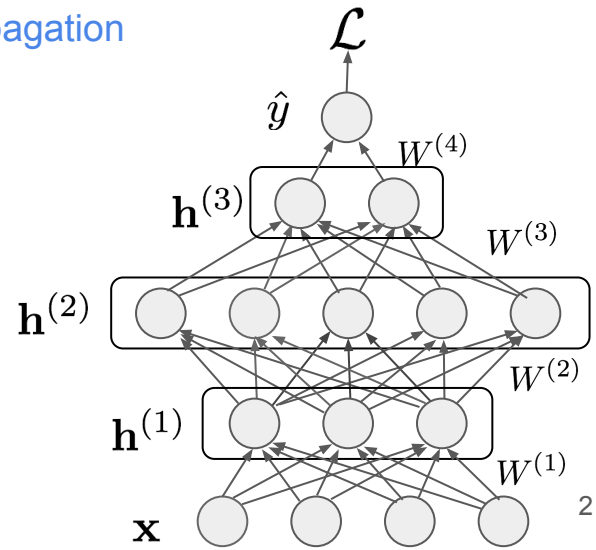
Learning fully connected multi-layer neural networks

- For a single data point, we can write the the hidden activations of the fully connected neural network as a recursive computation using the vector notation:

$$\mathbf{z}^{(1)} = W^{(1)}\mathbf{x} + b^{(1)}, \quad \mathbf{h}^{(1)} = \phi\left(\mathbf{z}^{(1)}\right)$$
$$\mathbf{z}^{(2)} = W^{(2)}\mathbf{h}^{(1)} + b^{(2)}, \quad \mathbf{h}^{(2)} = \phi\left(\mathbf{z}^{(2)}\right)$$
$$\mathbf{z}^{(3)} = W^{(3)}\mathbf{h}^{(2)} + b^{(3)}, \quad \mathbf{h}^{(3)} = \phi\left(\mathbf{z}^{(3)}\right)$$
$$\mathbf{z}^{(4)} = W^{(4)}\mathbf{h}^{(3)} + b^{(4)}, \quad \hat{y} = f\left(\mathbf{z}^{(4)}\right)$$

  - *f()* is the output activation function
  - The output of the network is then used to compute the loss function on the training data

forward propagation



Learning fully connected multi-layer neural networks

- For a single training data, computing the gradient w.r.t. the weight matrices is also a recursive procedure:
  - Remember: $\mathbf{z}^{(4)} = W^{(4)}\mathbf{h}^{(3)} + b^{(4)}, \quad \hat{y} = f\left(\mathbf{z}^{(4)}\right)$
  - Back-propagation is similar to running the neural network backwards using the transpose of the weight matrices

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(4)}} = \frac{\partial \hat{y}}{\partial \mathbf{z}^{(4)}}\frac{\partial \mathcal{L}}{\partial \hat{y}}, \quad \frac{\partial \mathcal{L}}{\partial W^{(4)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(4)}}\mathbf{h}^{(3)T}$$

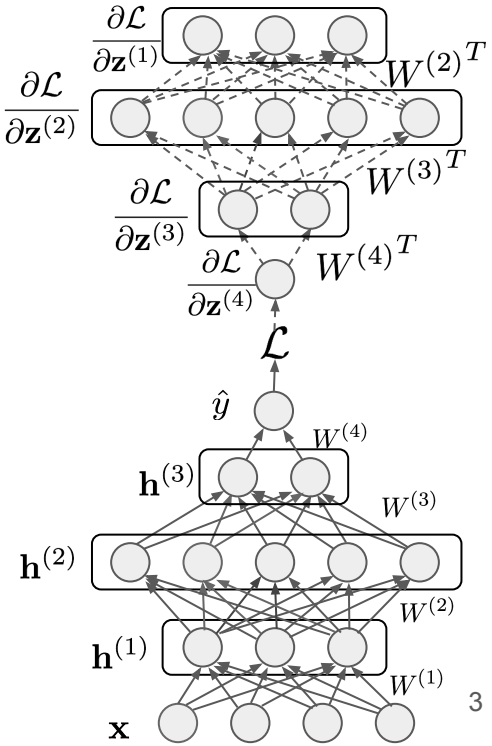
$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(3)}} = \frac{\partial \mathbf{h}^{(3)}}{\partial \mathbf{z}^{(3)}}\left(W^{(4)T}\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(4)}}\right), \quad \frac{\partial \mathcal{L}}{\partial W^{(3)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(3)}}\mathbf{h}^{(2)T}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(2)}} = \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{z}^{(2)}}\left(W^{(3)T}\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(3)}}\right), \quad \frac{\partial \mathcal{L}}{\partial W^{(2)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(2)}}\mathbf{h}^{(1)T}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}} = \frac{\partial \mathbf{h}^{(1)}}{\partial \mathbf{z}^{(1)}}\left(W^{(2)T}\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(2)}}\right), \quad \frac{\partial \mathcal{L}}{\partial W^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}}\mathbf{x}^{T}$$

back-propagation

What about the expression for the bias units?

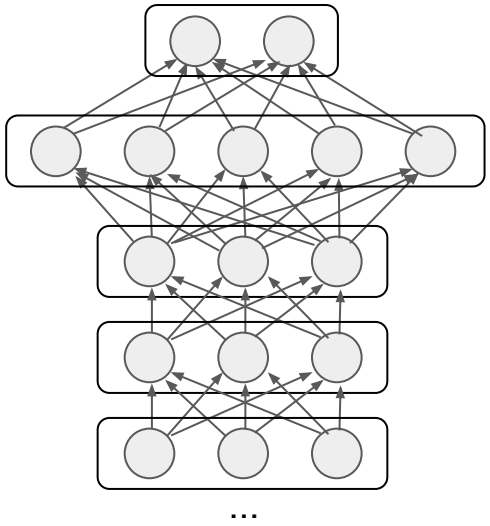


Outline

- **Bag-of-tricks for deep neural networks**
  - Local minimums and initialization
  - Early stopping and regularization
  - Dataset normalization
- Type of neural networks
  - Building blocks
  - Convolutional neural network
  - Recurrent neural network

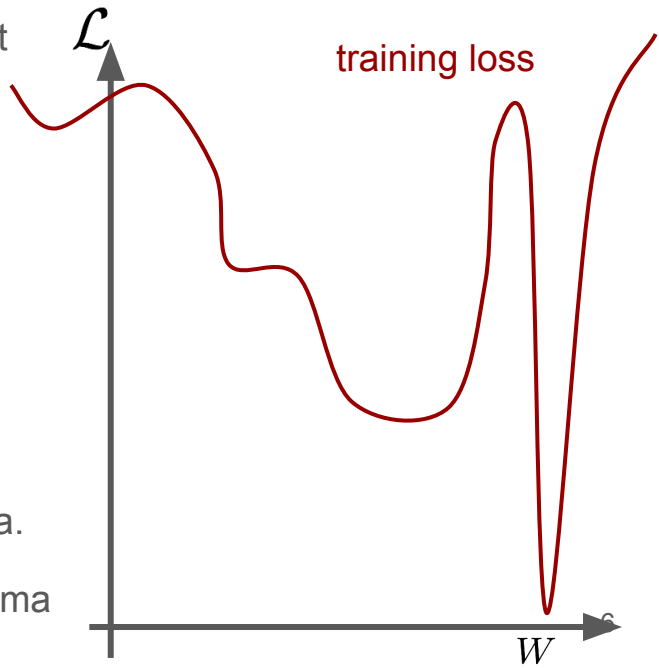
Hyper-parameters

- At the beginning, there were the choices:
  - How many hidden units to use in each hidden layer?
  - How many layers in total?
  - Which hidden activation function?
- Good answer: decide these hyper-parameters using validation set.
- Best practical answer:
  - Around 500-2000 hidden units
  - 2-3 layers
  - Choosing ReLU often leads to fast convergence



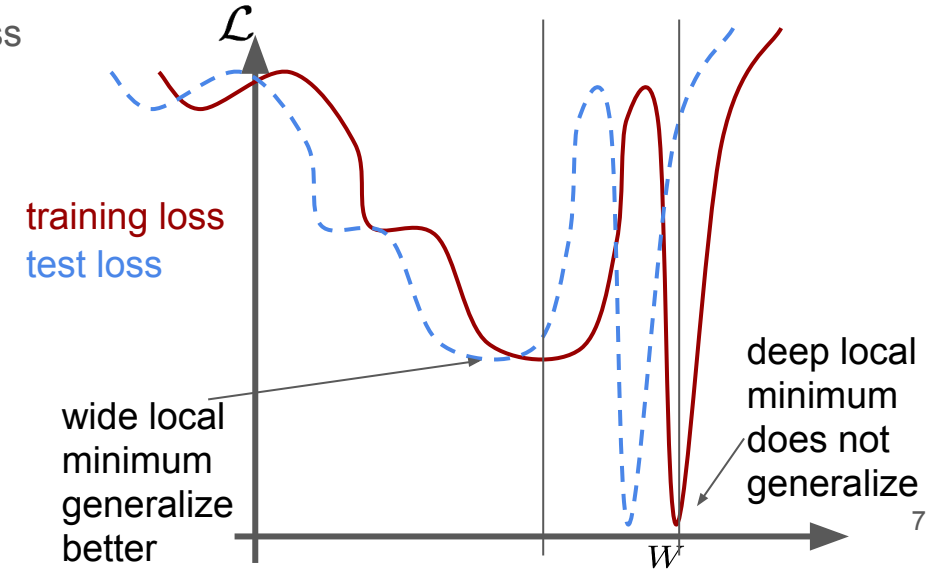
Parameter initialization

- The loss functions for neural networks w.r.t. the weight matrices in general are non-convex.
  - Different weight-initialization schemes can lead to significant differences in final performance.
  - Use random initialization (e.g. Gaussian with std. 0.01) and avoid constant initialization
- Non-convex optimization is not crazy. Most of the local minima in a neural network's loss function are not bad.
  - As long as the model has enough capacity to model the data.
  - Stochastic gradient descent can usually find good local minima



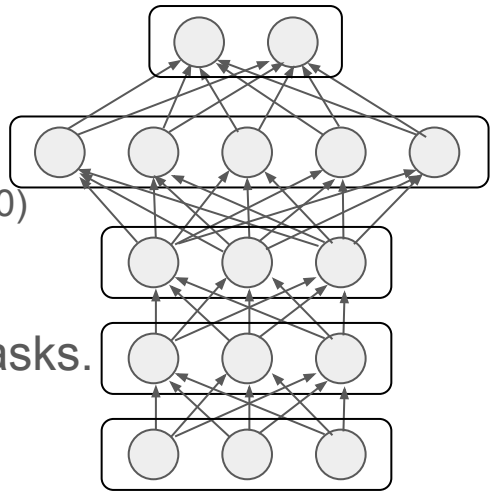
Some local optima generalize better than others

- Consider two neural networks that each achieve a low error rate on the training set. We prefer the model that learns the underlying statistical patterns of the data and can generalize to unseen examples during test time.
  - The test loss function and the training loss are almost always slightly different.
  - Wide shallow basins can typically generalize better than deep narrow local minima.
  - The subsampling noise from SGD helps to find shallow basins.



Careful initialization

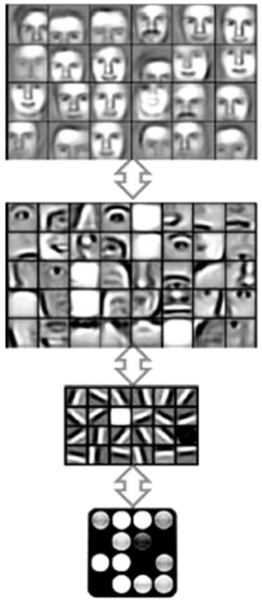
- For really deep neural networks (> 5 layers), random initialization from a constant-variance Gaussian noise will not work well. The back-propagated partial derivatives will likely be too small to learn anything useful.
  - Simple fix: initialize the weight matrices to identity matrices if you can. (Le, Jaitly and Hinton, 2015)
  - More elaborate fix: adapt the std. of the zero-mean Gaussian initialization to be sqrt{3/{#inputs + #outputs}} aka *Xavier initialization* (Xavier Glorot and Yoshua Bengio, 2010)
- Oftentimes it is beneficial to initialize the weights from another model that was “pre-trained” on some other tasks.
  - Initialize the model from an auto-encoder or ImageNet models.
  - Weight matrices in the early layers are transferable and help generalization. …



Pre-training and deep learning hypothesis

- One of the hypothesis why deep learning works so well is that deep nets learn appropriate low, mid and high level features given enough data.

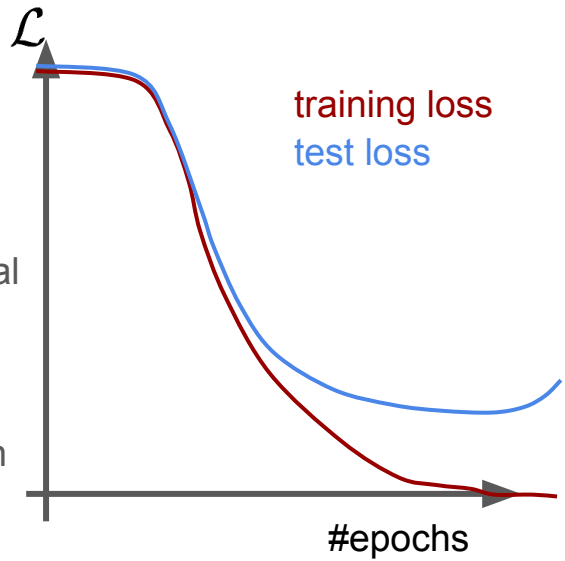
- Lower level layers learn local but general feature detectors.
  - It makes sense to transfer the early hidden layers and expect those low level features to work well
  - If we do not have enough data for the current task, we can expect a win by training models on a similar but not identical task with millions of training examples and then transfer the model back to the current task



[from Andrew Ng's slides]

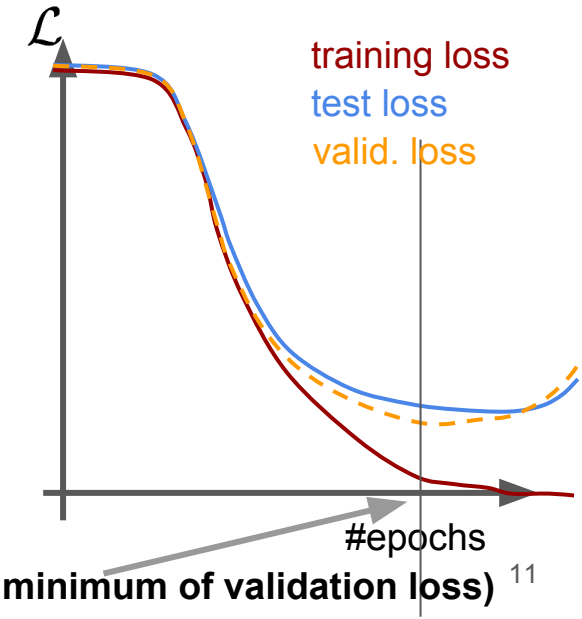
Regularize the capacity of deep neural networks

- Given enough hidden units, neural networks can overfit to the training dataset. Regularizing a neural network is equivalent to restricting its capacity.
  - Reducing the number of hidden units to limit the model capacity has a sound statistical justification: you need more training examples than the number of weights to have enough *statistical power* when estimating the unknown model parameter.
  - Alternatively, one can have an over-parameterized model and deal with overfitting through a very strong regularizer such that most of the weights are close to zero, e.g. using weight decay.
  - In deep learning applications, strong-regularizer approaches often work much better. A good heuristic is to prefer a globally simple prediction function with some irregular local structure.



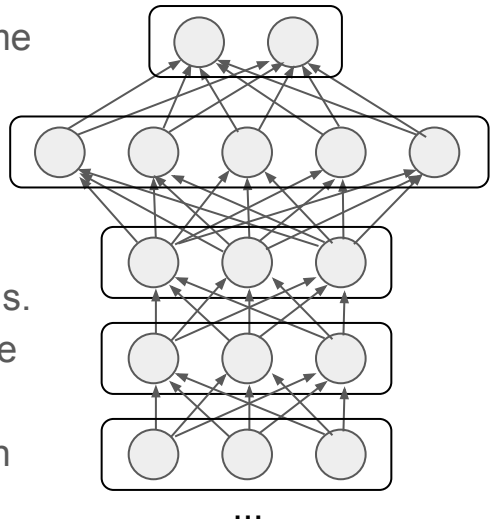
Early stopping

- A very simple trick to ensure that most of the weights are as close to zero as possible involves monitoring the *validation loss.* You stop learning at the minimum of the validation loss curve (aka “early stopping”) before attaining the minimum training loss.
    - The goal of machine learning is to generalize well on test data instead of finding the minimum training loss.
    - The weights are typically initialized around zero. “Early stopping” terminates the learning process prematurely before the weights grow too large and thus limits the capacity of the model.
    - It is by far the most commonly used regularization technique.



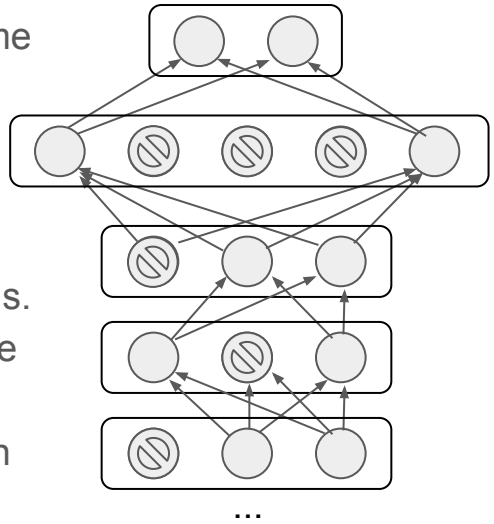
Dropout

- Another simple trick to effectively regularize deep NNs is to remove hidden units randomly during training. Dropout prevents “co-adaptation” of the hidden units and encourages independence among the neurons.
  - Dropout can be understood as stochastic training on all the permutations of neural network architectures that share the same weight matrices.
  - Efficiently shares the statistical power among an ensemble of neural networks.
  - During test time, the “mean” network is used to make predictions. If each hidden unit is dropped out 50% of the time in training, we need to compensate during test time by *reducing* the weight matrix by a factor of 2, such that the expected activation of each hidden unit stays the same for both training and testing.



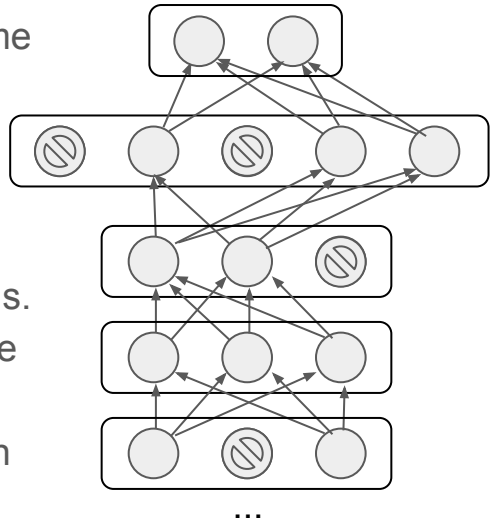
Dropout

- Another simple trick to effectively regularize deep NNs is to remove hidden units randomly during training. Dropout prevents “co-adaptation” of the hidden units and encourages independence among the neurons.
  - Dropout can be understood as stochastic training on all the permutations of neural network architectures that share the same weight matrices.
  - Efficiently shares the statistical power among an ensemble of neural networks.
  - During test time, the “mean” network is used to make predictions. If each hidden unit is dropped out 50% of the time in training, we need to compensate during test time by *reducing* the weight matrix by a factor of 2, such that the expected activation of each hidden unit stays the same for both training and testing.



Dropout

- Another simple trick to effectively regularize deep NNs is to remove hidden units randomly during training. Dropout prevents “co-adaptation” of the hidden units and encourages independence among the neurons.
  - Dropout can be understood as stochastic training on all the permutations of neural network architectures that share the same weight matrices.
  - Efficiently shares the statistical power among an ensemble of neural networks.
  - During test time, the “mean” network is used to make predictions. If each hidden unit is dropped out 50% of the time in training, we need to compensate during test time by *reducing* the weight matrix by a factor of 2, such that the expected activation of each hidden unit stays the same for both training and testing.



Statistical power of the MLE v.s. neural networks

- Traditional view of machine learning is that we need more training examples than the number of parameters.
  - E.g. fitting a linear regression of 100 inputs on thousands of training examples.
  - E.g. fitting the hyper-parameters (weight decay) on hundreds of validation examples.
- Neural networks typically have millions of learnable weights and there are always not enough data. (we would just keep building fancier and larger models once we had more training examples)
  - It is almost always better off to over-parameterize and then prevent overfitting by applying very strong regularizer. (e.g. dropout)
  - Or to transfer statistical efficiency from other pre-trained models.

Summary of regularization options

- Lower the number of units and depth
  - Often due to computational reasons
- Weight decay
  - Should always have a small amount of weight decay
- Early stopping
  - Very effective, always apply early stopping
- Dropout
  - Strength of the regularization can be increased or decreased by the dropout probability
- Fine-tuning pre-trained models
  - Initialize your network from some model pre-trained on millions of data points

Dataset normalization

- Learning can often be made easier by pre-processing the dataset before running the learning algorithms.
  - The simplest normalization scheme is to centre the input $\mathbf{x}$ and remove its variance for each input dimension. This fixes the scaling discrepancy among the input dimensions (e.g. removes the units of measurement). SGD learning works much better on the inputs that have the same scale. **Why**?
  - For each of the *N* input dimensions, estimate its mean and its variance from the training data. Then normalize all the training example:

$$\bar{x}_n^{(m)} = (x_n^{(m)} - \mu_n)/\sigma_n$$

$$\mu_n = \frac{1}{M}\sum_{m=1}^{M} x_n^{(m)} \qquad \sigma_n = \sqrt{\frac{1}{M-1}\sum_{m=1}^{M}(x_n^{(m)} - \mu_n)^2}$$

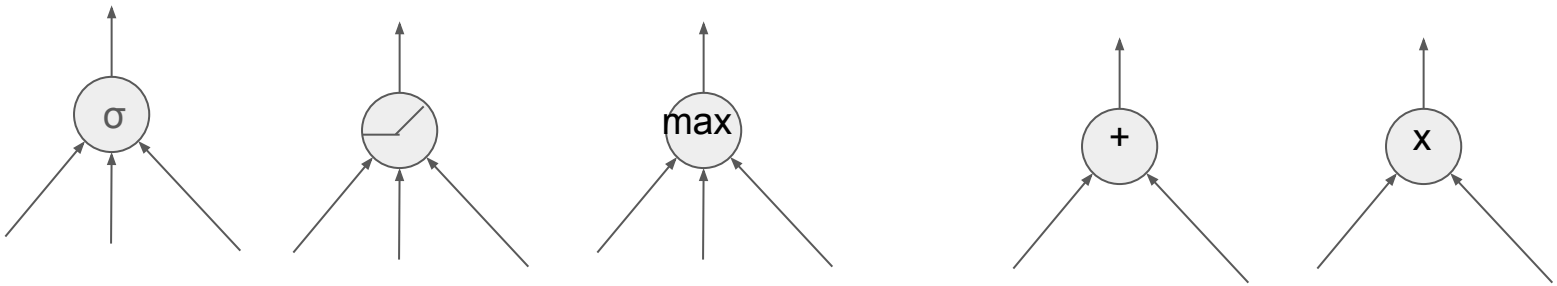
  - We can further remove the covariance among the input dimensions, i.e. whitening the data using Principal Component Analysis (PCA): next lecture

Outline

- Bag-of-tricks for deep neural networks
  - Local minimums and initialization
  - Early stopping and regularization
  - Dataset normalization
- **Type of neural networks**
  - Building blocks
  - Convolutional neural network
  - Recurrent neural network

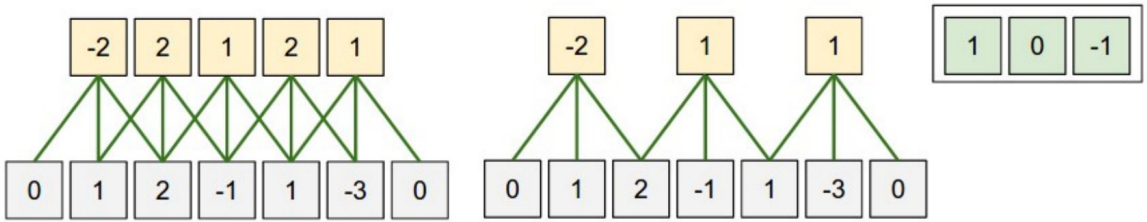
Basic neural building blocks

- Here is some basic computational units that we will use to construct deep neural networks with more advanced connectivity patterns.
  - All these basic computational units have well defined partial derivatives.



Sparse local connectivity

- When keeping the general stacked layer-wise neural network architecture, we can have specialized connectivity pattern between the hidden layers:

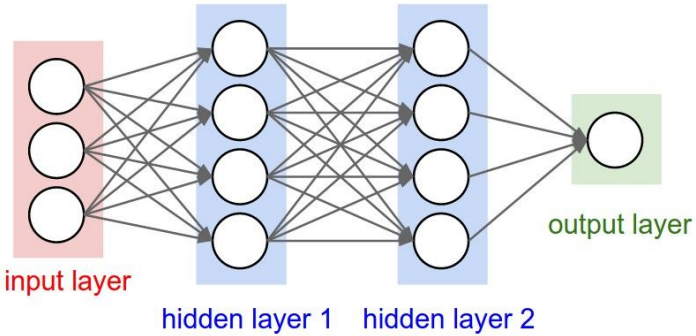


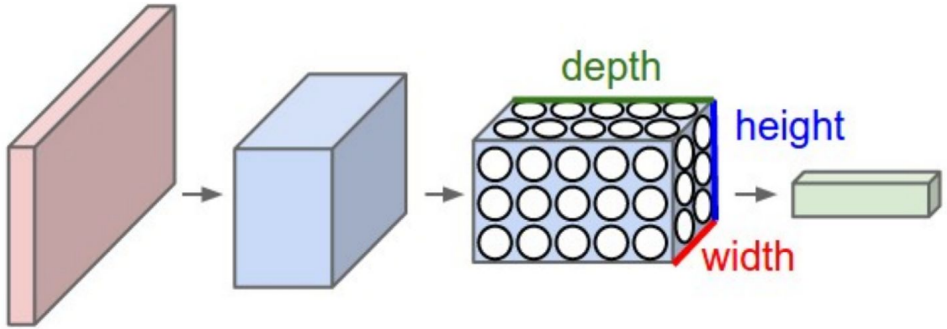
  - **Local connectivity**: each hidden unit operates on a local image patch (3 instead of 7 connections per hidden unit)
  - **Weight sharing**: filtering each chunk of the signal the same way. (e.g. processing each patch of the image the same way.) Parameter sharing improves statistical power and reduces overfitting.
  - Now we can have a lot more neurons: # neurons vs. # weights. It works well to match a pattern in image

[from Piercy Liang's slides and Andrej Karpathy's blog]

ConvNet: 2D convolution

- Convolutional architecture where each depth column is produced from localized region (in height/width):

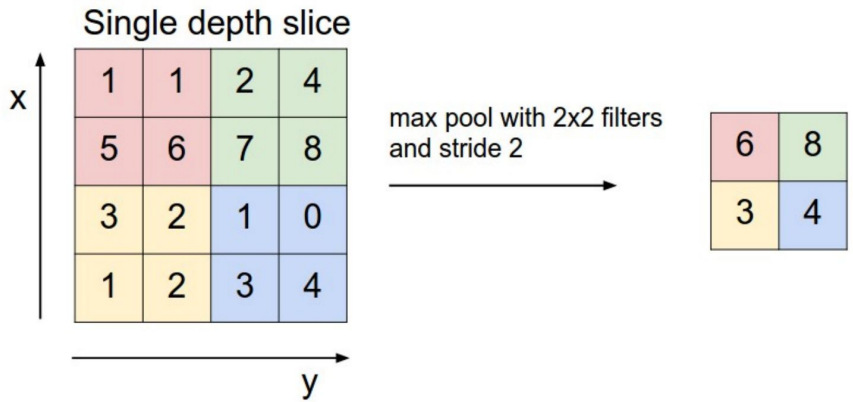




[from Piercy Liang's slides and Andrej Karpathy's blog]

ConvNet: Max-pooling

- We can reducing the dimension of the input without introducing extra parameters by “pooling” the maximum value of a neighbourhood:

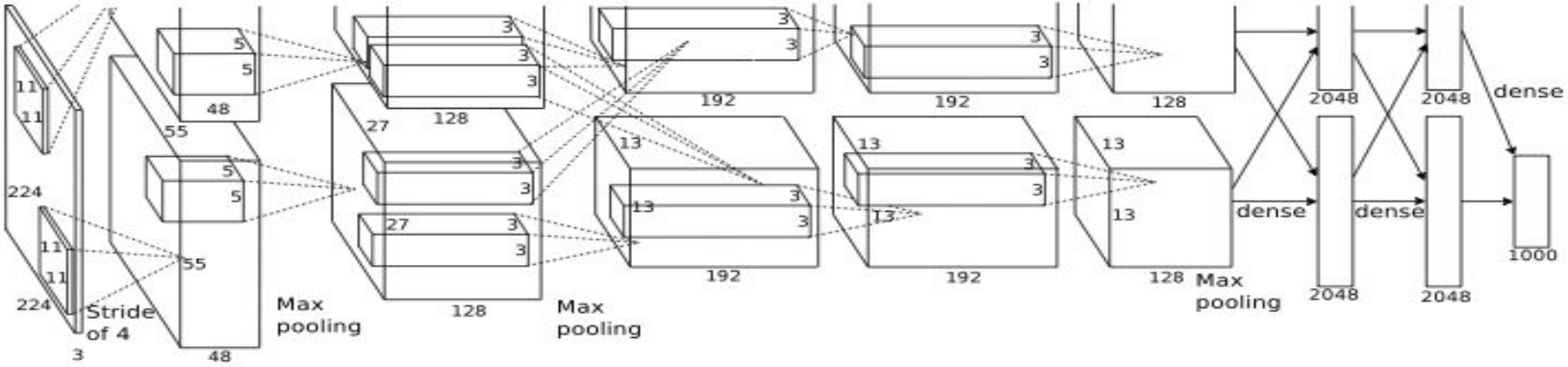


  - The pooling layer tests if there is a strong pattern in a local neighbourhood while suppressing others
  - Max-pooling layer also prevent overfitting because it has not learnable parameters

[from Piercy Liang’s slides and Andrej Karpathy’s blog]

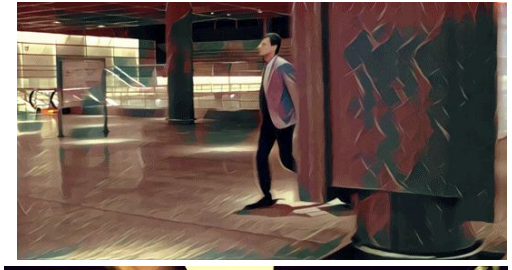
CNN: object recognition

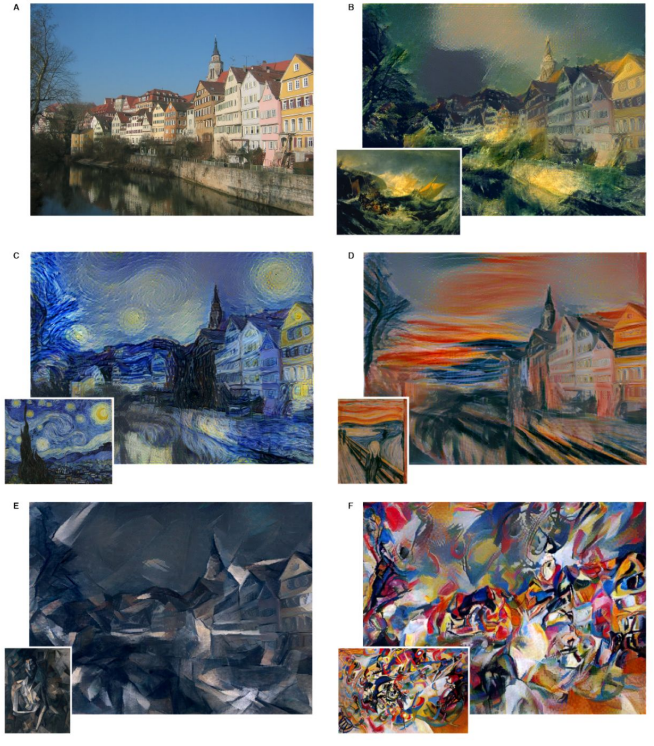
- AlexNet: a typical modern CNN

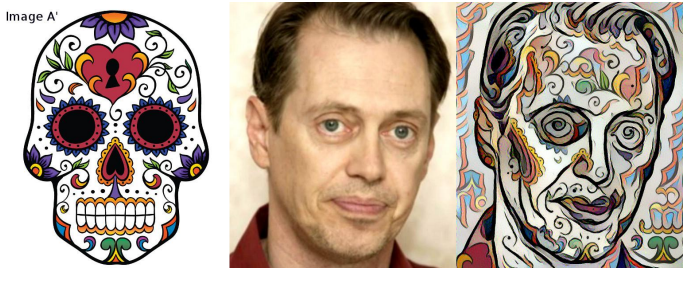


[Krizhevsky, et. al., 2012]

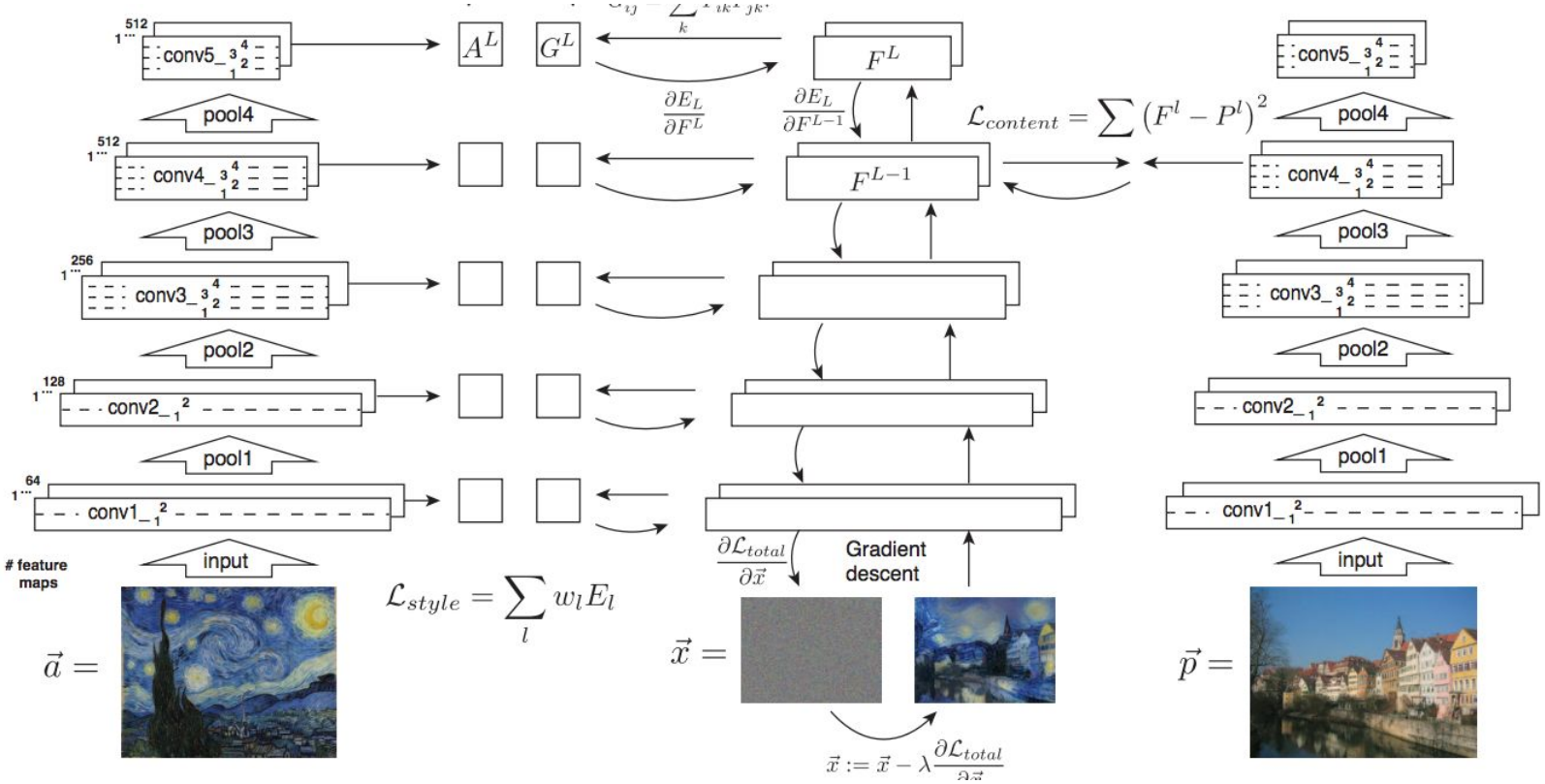
CNN: neural style transfer







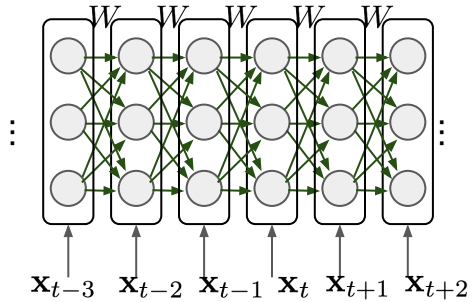
CNN: neural style transfer



[Gatys, et. al., 2015]

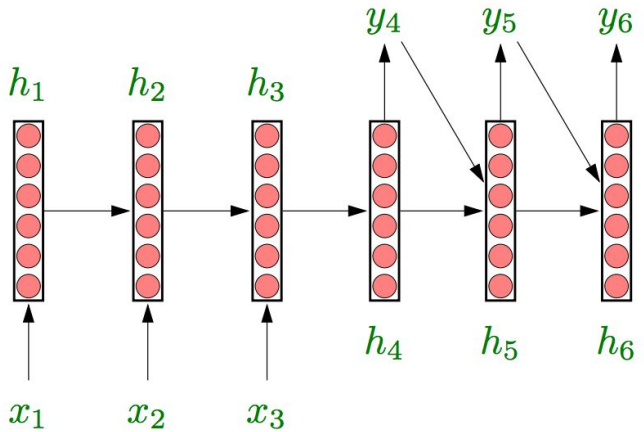
Recurrent neural network

- Weight sharing is a general technique that can be applied to any neural networks.
    - In CNNs, weight sharing is applied to the neurons in the same hidden convolutional layer. Here, we apply weight sharing among weight matrices. The same weight matrix “recurs” between the hidden layers.
    - Sharing the same weight matrix in a very deep fully connected neural network allow us to build an autoregressive process: a process in which future values are estimated based on past values estimated using the same model. An autoregressive process operates under the premise that past values have an effect on current values.



RNN: statistical machine translation

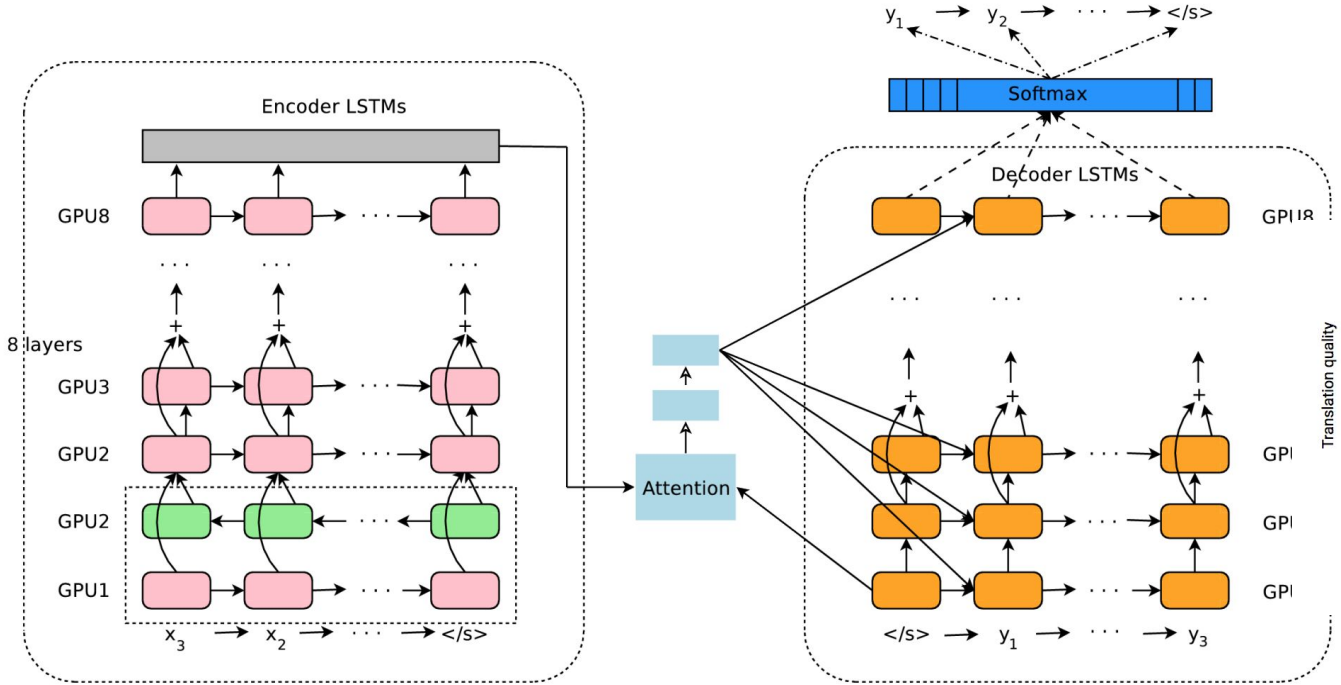
- RNN can deal with variable length input sequence, i.e. the inputs to RNNs does not have to be a fixed length vector anymore.
- Machine translation:
  - x:Je crains l'homme de un seul livre.
  - y:Fear the man of one book.

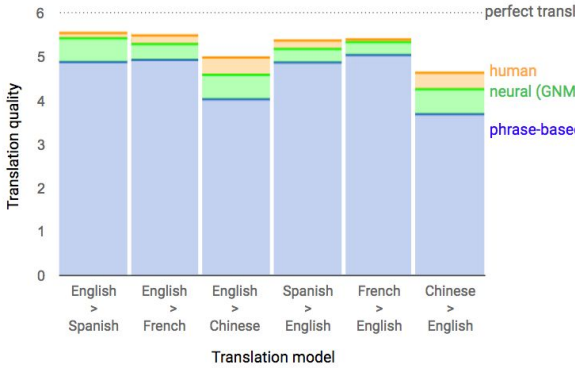


[from Piercy Liang's slides and Andrej Karpathy's blog]

RNN: statistical machine translation

- Large scale Google Neural Machine Translation: deep 8 layer RNNs

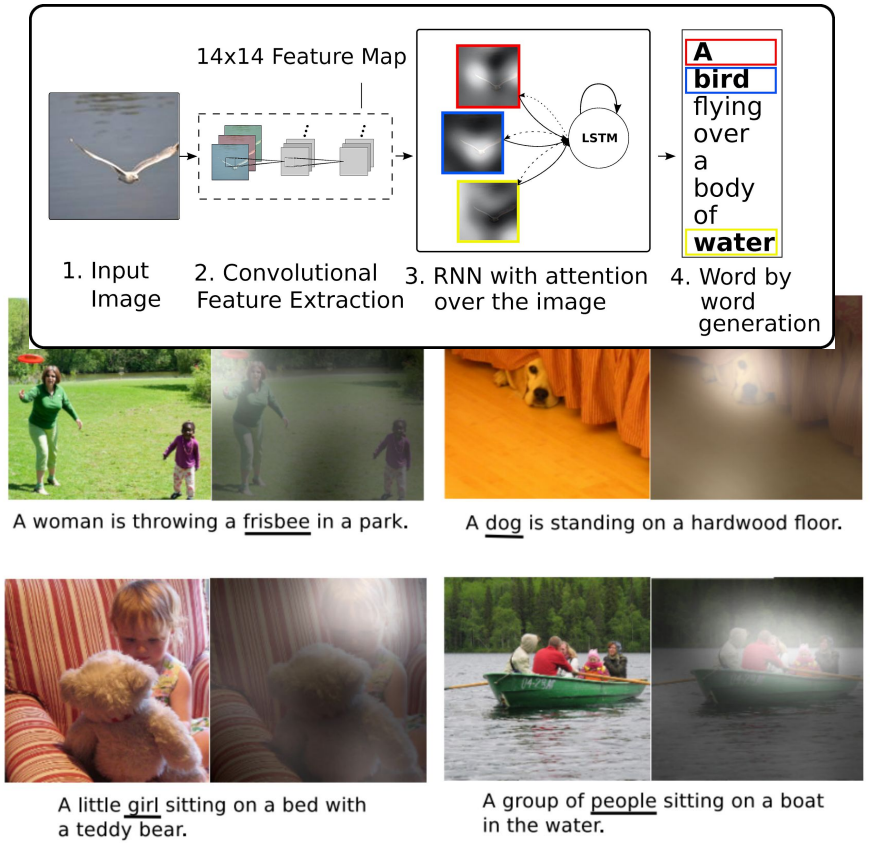




[Wu, et. al., 2016]

Combining CNN with RNN: Caption generation

- The model consists of a convolutional network that consume a fixed size input vector (a fixed size image). The last hidden layer of the CNN is connected to the RNN that outputs a sequence of words to describe the input image.



[from Xu et.al. 2014]