CS 505: Introduction to Natural Language Processing

Wayne Snyder
Boston University

Lecture 17 – Neural Network Tuning and Advanced Features, Part 2

(Model Design + Hyperparameters) → Model Parameters

The building blocks:
- # Layers
- Activations
- Optimizers
...

The knobs that you can turn:
- Learning Rate
- Dropout
...

The variables learned from the data:
- weights
...

Lecture Plan

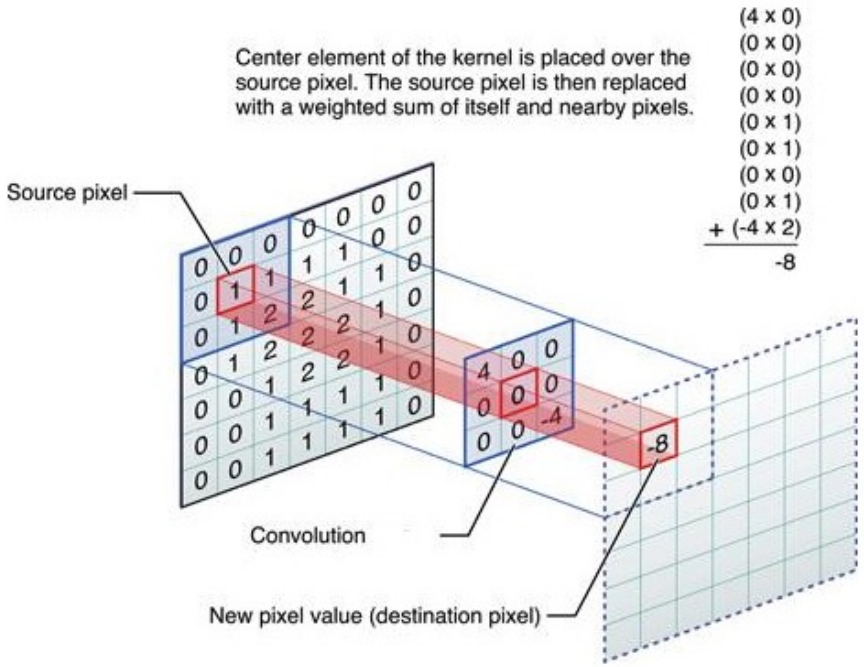
Best Practices and Advanced Features of Neural Networks for NLP

- More types of layers:
  - Convolutional & Pooling
  - Embedding
- Network architectures for NLP
  - How deep?
  - How wide?

Convolutional Neural Networks

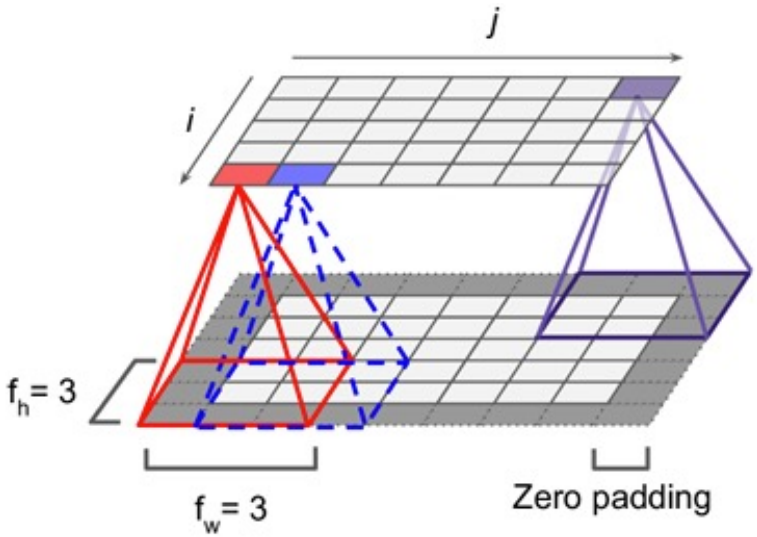
Convolutional nueral networks have had great success in image processing, and to a more limited degree, in NLP.

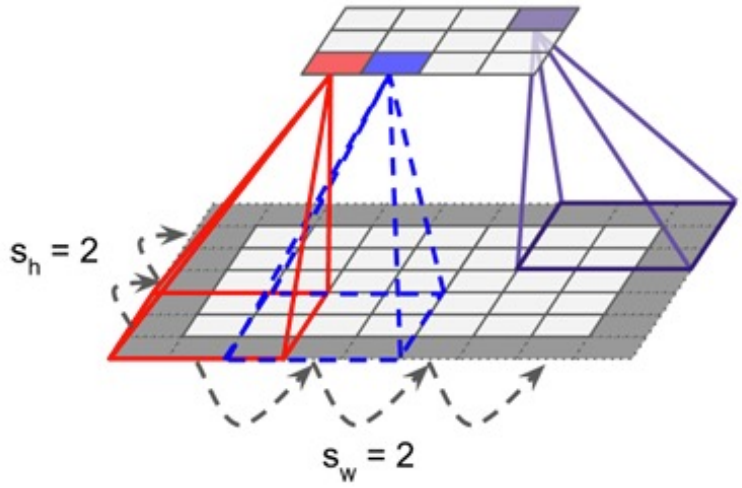
CNNs add convolution and pooling layers to focus on small regions of the data (here, images). Each input to the next layer is calculated as the dot product of the convolution kernel with a small region of the image.



Convolutional Neural Networks

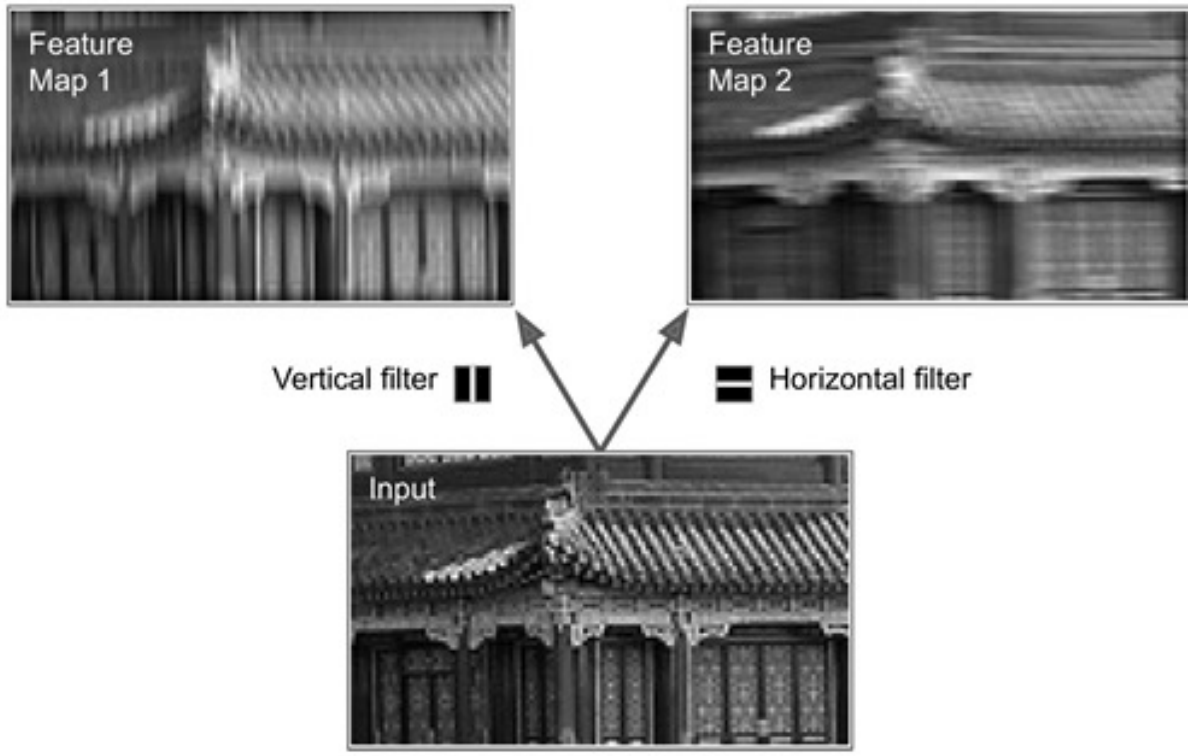
The convolution kernels are moved around the image, perhaps by some skip....





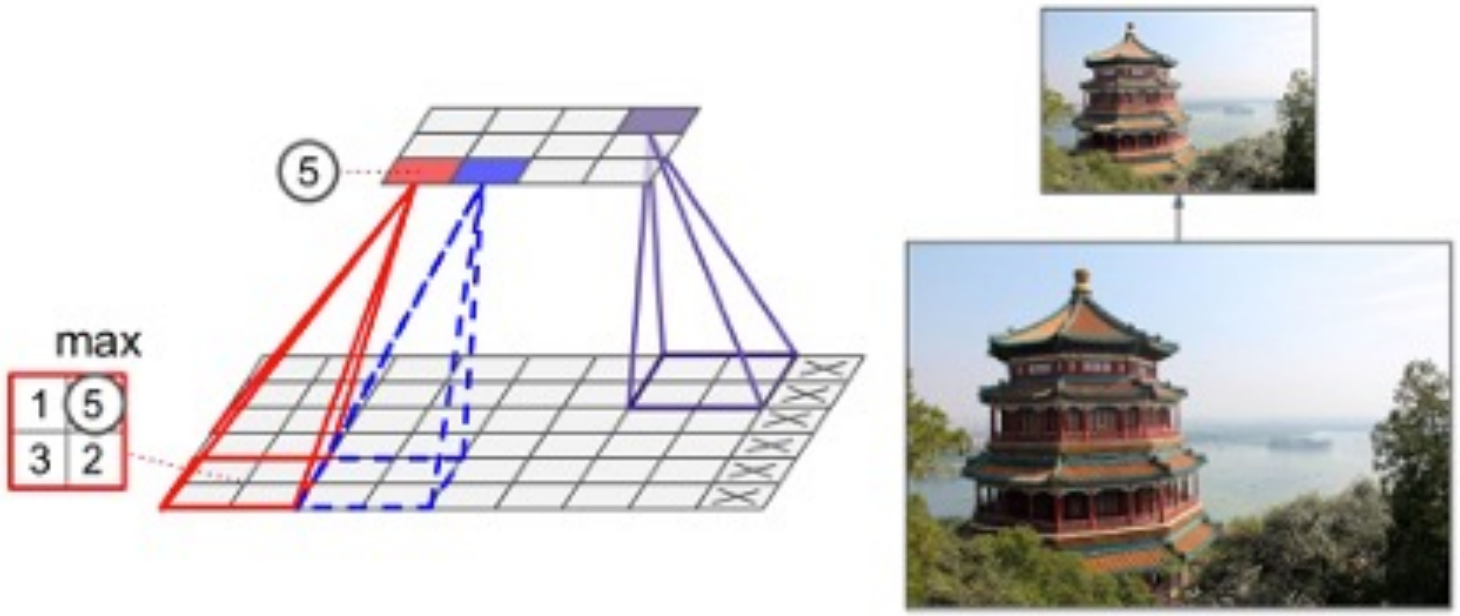
Convolutional Neural Networks

Since data is shared in the region covered by the kernel, various "features" of the image can be recognized in multiple places around the image:



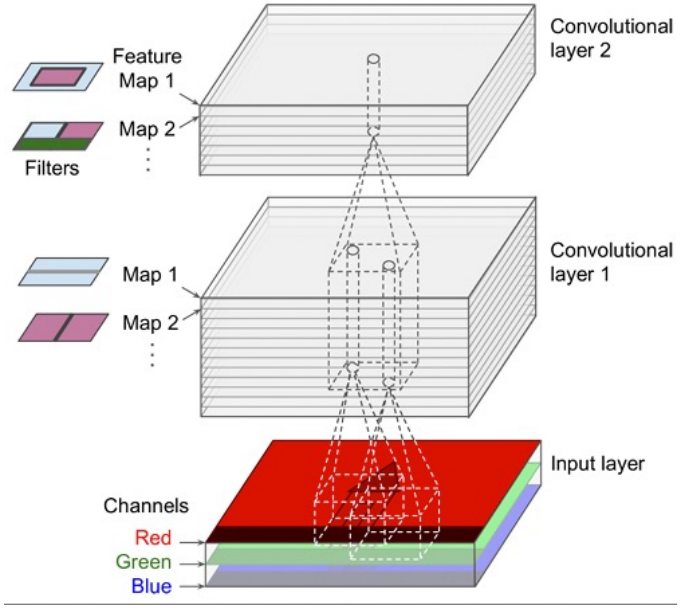
Convolutional Neural Networks

A pooling layer performs dimensionality reduction by averaging (or taking the maximum) of small regions in the previous layer:



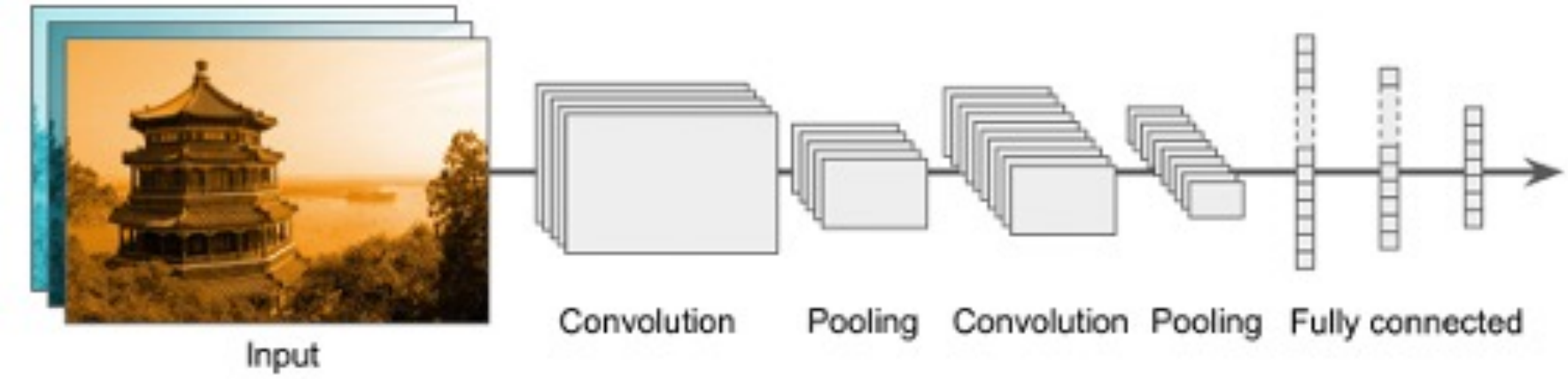
Convolutional Neural Networks

And layers can have multiple “maps” and be stacked:



Convolutional Neural Networks

It is typical to alternate convolution and pooling layers and end with fully connected layers before output:



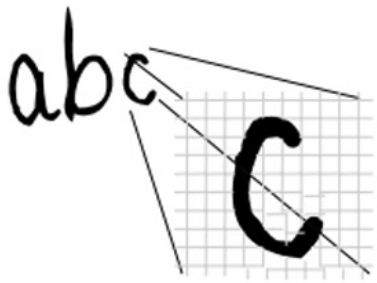
Convolutional Neural Networks

So… if this is an NLP class, why are we discussing CNNs for images??

Three reasons:

One: Language is not just represented by linear sequences of Unicode symbols, it also exists in handwritten form!

In fact, the very first big success in NNs was achieved by Yann LeCunn and colleagues at Bell Labs in the 1990s, solving the problem of handwritten digit and letter recognition using CNNs:



Off-line

Gradient-Based Learning Applied to Document Recognition

YANN LECUN, MEMBER, IEEE, LÉON BOTTOU, YOSHUA BENGIO, AND PATRICK HAFFNER

Invited Paper

Table 14-1. LeNet-5 architecture

Layer	Type	Maps	Size	Kernel size	Stride	Activation
Out	Fully Connected	–	10	–	–	RBF
F6	Fully Connected	–	84	–	–	tanh
C5	Convolution	120	1 × 1	5 × 5	1	tanh
S4	Avg Pooling	16	5 × 5	2 × 2	2	tanh
C3	Convolution	16	10 × 10	5 × 5	1	tanh
S2	Avg Pooling	6	14 × 14	2 × 2	2	tanh
C1	Convolution	6	28 × 28	5 × 5	1	tanh
In	Input	1	32 × 32	–	–	–

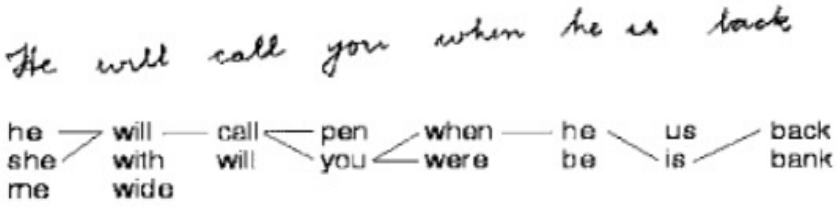
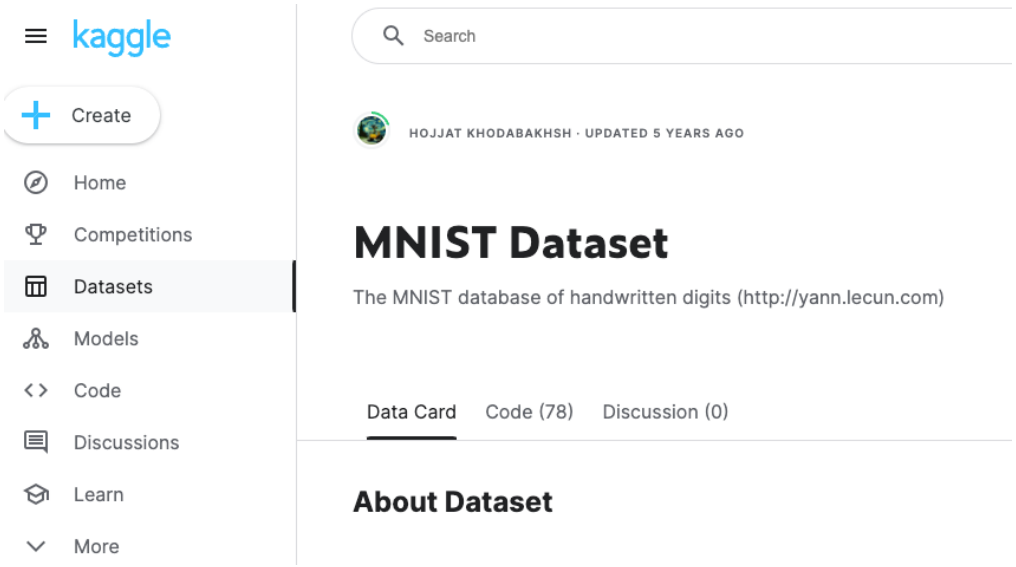


Figure.3. Recognition of a line

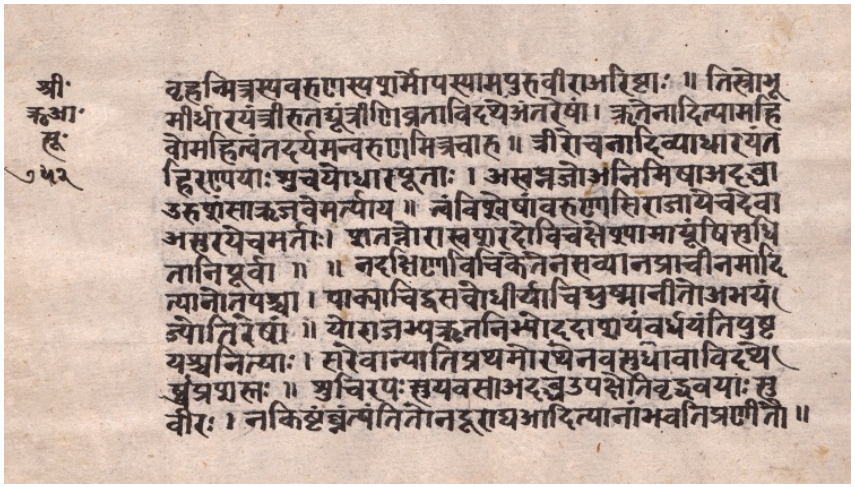
Convolutional Neural Networks

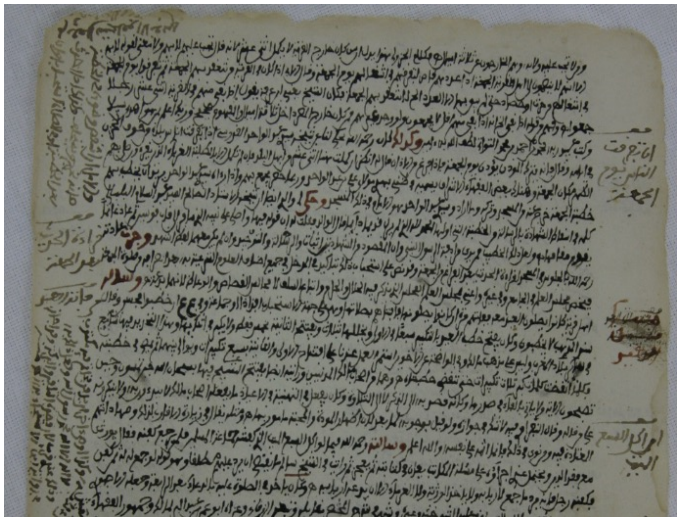
The dataset from that classic paper is now the “Hello World” of neural network programming!

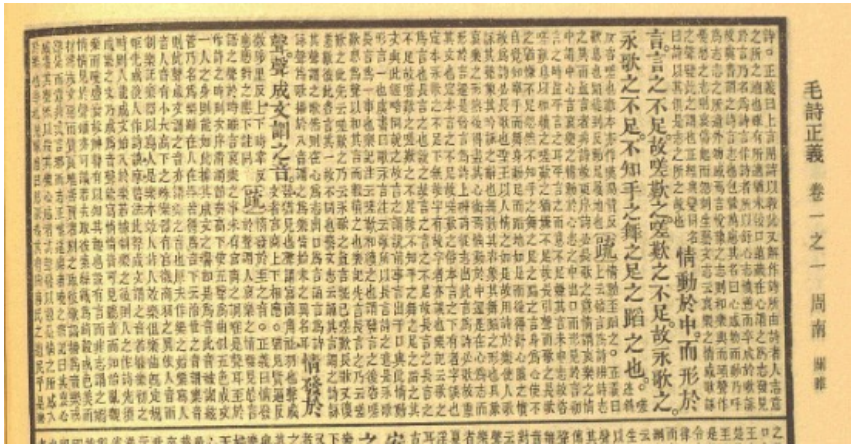


Convolutional Neural Networks

Similar techniques are used in decoding and transcribing historical manuscripts (handwritten many centuries before Unicode!):

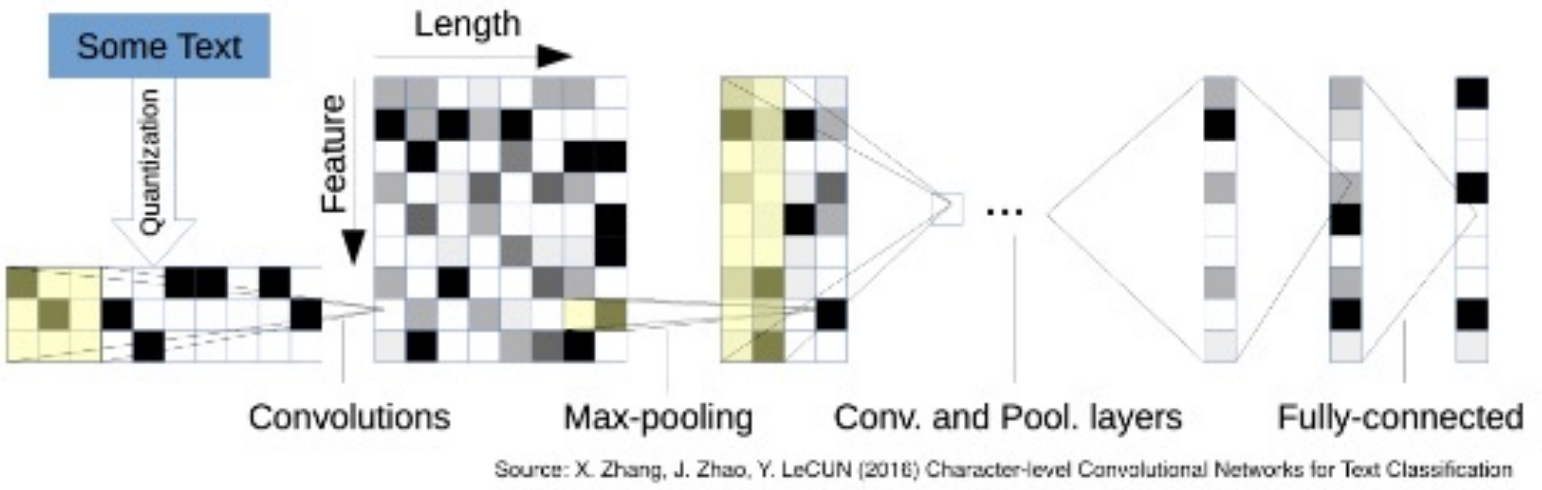




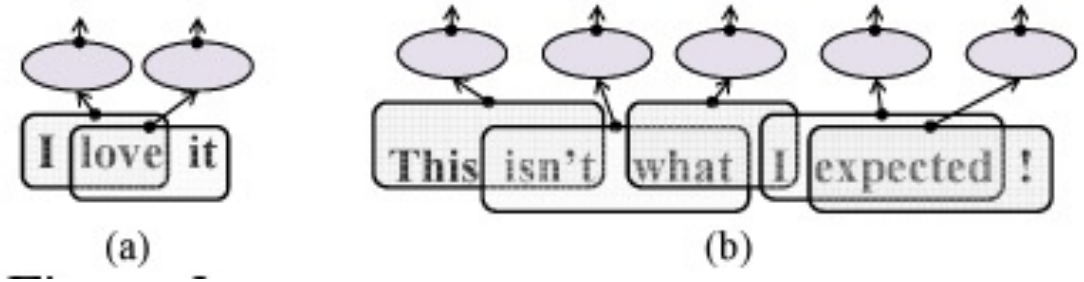


Convolutional Neural Networks

Second: 2D CNNs turned out to not be that useful (say in matrices of embedding sequences), but 1D CNNs (which essentially recognize N-grams) have proved useful in sentiment analysis – essentially the convolution recognizes N-grams!



Source: X. Zhang, J. Zhao, Y. LeCUN (2016) Character-level Convolutional Networks for Text Classification



Convolutional Neural Networks

Third: NLP techniques based on CNNs are used in many non-linguistic contexts, particularly in analyzing DNA and other linear molecules:

Check for updates

OPEN

Improving protein succinylation sites prediction using embeddings from protein language model

Suresh Pokharel[1], Pawel Pratyush[1], Michael Heinzinger[2,3], Robert H. Newman[4,5] & Dukka B. KC[1]✉

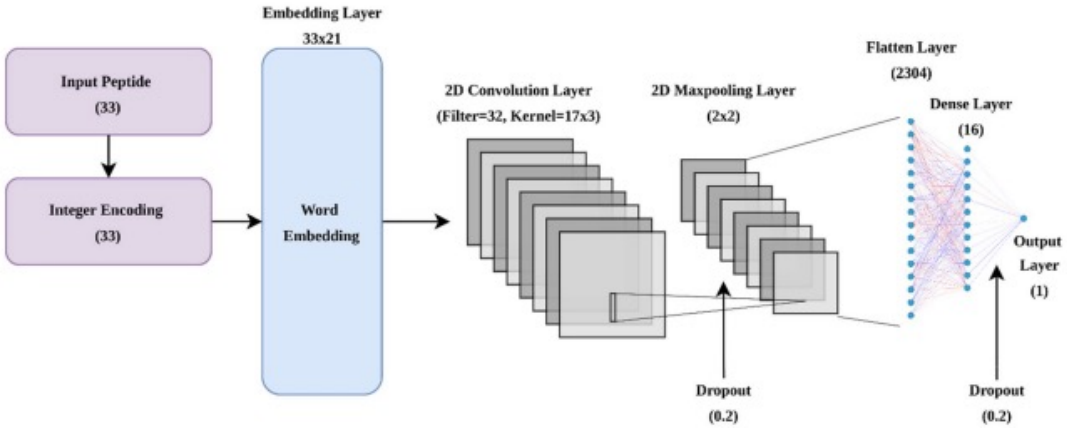


Figure 1. Architecture of supervised word embedding based model using a convolutional neural network.

Convolutional Neural Networks

CNNs in Pytorch:

```
class CNN_Net1(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3)
        self.conv2_drop = nn.Dropout2d(0.4)
        self.fc2 = nn.Linear(500, 100)
        self.fc3 = nn.Linear(100, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(F.max_pool2d(x, 2))
        x = self.conv2(x)
        x = self.conv2_drop(x)
        x = F.relu(F.max_pool2d(x, 2))
        x = x.view(-1, 1600)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        x = self.fc3(x)
        return x
```

Embedding Layers

Current NLP systems all use word embeddings as inputs:

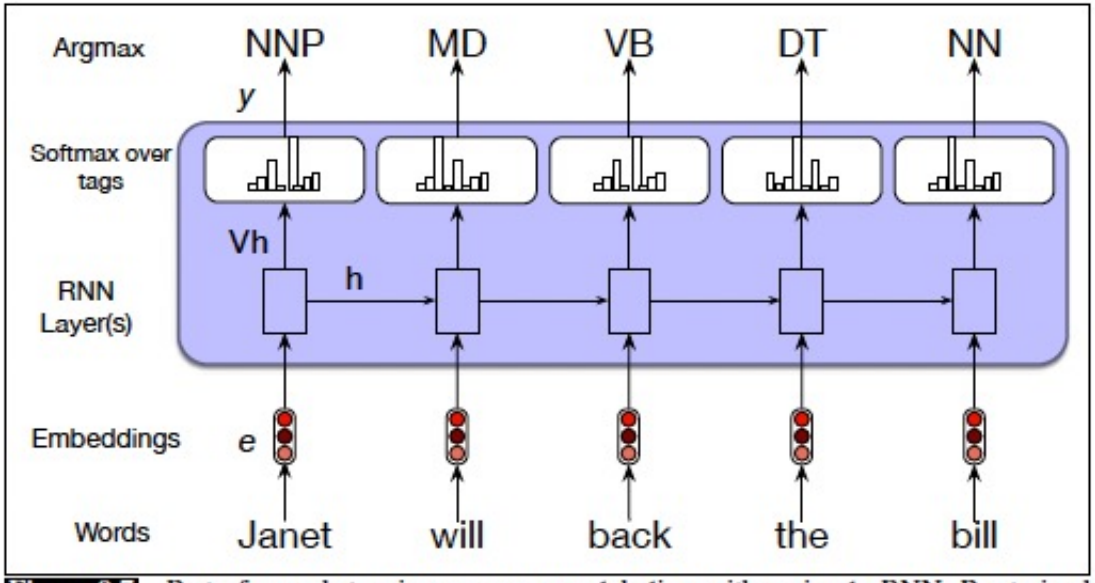


Figure 9.7 Part-of-speech tagging as sequence labeling with a simple RNN. Pre-trained word embeddings serve as inputs and a softmax layer provides a probability distribution over the part-of-speech tags as output at each time step.

Therefore, Pytorch and other NN platforms provide built-in embedding layers….

Embedding Layers

An embedding layer is a simple lookup table which maps indices to embedding vectors (an array of D floats for some dimension D).

By default, the embedding layer is untrained, so the vectors are random floats:

In [19]:
```
class EmbeddingExample(nn.Module):

        def __init__(self):
            super().__init__()
            self.embedding_layer = nn.Embedding(10,50)

        def forward(self,x):
            x = self.embedding_layer(x)
            return x

embedding_model = EmbeddingExample()

embedding_model( torch.tensor(0) )
```

Out[19]:
```
tensor([-1.5466, -0.7189, -1.3827, -0.1787,  1.0937,  1.8469,  0.3557,  1.8965,
         1.1616, -1.3141, -0.7681,  0.1190, -0.5923, -0.2816,  0.9321,  1.2912,
        -1.0337, -0.4085,  0.8838,  0.4950, -0.4681,  1.2889, -1.0458,  1.2371,
        -0.7286, -1.3844, -0.5985, -0.8621, -1.8770, -3.3333, -0.2535, -0.0394,
        -0.5554, -0.1743,  0.7165,  0.3712,  1.1531,  0.9583,  1.9512,  0.3636,
         0.9459, -2.3356,  0.6509,  1.3928, -1.9889,  0.0838,  0.2376,  0.0719,
        -1.6311, -0.0457], grad_fn=<EmbeddingBackward0>)
```

Embedding Layers

The vocabulary is represented by (tensor) indices from into the vocabulary array:

In [20]:
```
1 sentence = "This is a sentence of words for the embedding example"
2
3 words = sentence.lower().split()
4
5 vocab = sorted(set(words))
6
7 vocab_idx = { w : torch.tensor(i) for (i,w) in list(enumerate(vocab)) }
8
9 vocab_idx
```

Out[20]:
```
{'a': tensor(0),
 'embedding': tensor(1),
 'example': tensor(2),
 'for': tensor(3),
 'is': tensor(4),
 'of': tensor(5),
 'sentence': tensor(6),
 'the': tensor(7),
 'this': tensor(8),
 'words': tensor(9)}
```

In [21]:
```
1 embedding_model( vocab_idx['sentence'] )
```

Out[21]:
```
tensor([-1.0522, -1.5031, -0.8273, -0.2849, -0.6338,  0.9456, -0.1177,  0.4843,
        -1.0190,  0.4294, -0.6801, -0.7463,  2.4668, -2.3485,  0.1480,  0.5330,
         0.3156, -1.5070, -0.2272,  1.5661, -1.4879, -0.0688,  0.4833,  0.7480,
        -1.4878, -0.4362,  0.4736, -0.5361, -0.0888, -0.4081, -0.4166,  0.2677,
        -0.4611, -1.4399, -0.4965,  0.4376,  2.4829, -0.4196,  1.4912,  0.2124,
         0.1241,  0.4749, -0.3347, -0.2382,  0.2249,  0.5873,  1.4045,  0.9617,
        -0.6000, -0.4867], grad_fn=<EmbeddingBackward0>)
```

Embedding Layers

Embedding Layers can be used in two (overlapping) ways:

- You can create randomly-initialized embeddings, and train them via backprop as you process sequences of words in your task.
- You can load pretrained embeddings such a GloVe and use them as is; or
- You can do both: import pretrained embeddings and further train them to specialize to your corpus.

Embedding Layers

In Pytorch using torchtext:

The only complication is that for each word in our vocabulary, we have to replace the initial random weights with the GloVe embedding if it is available, else initialize randomly.

```
# Load pre-trained GloVe embeddings
def load_glove_embeddings(path):
    ….
embeddings = load_glove_embeddings('glove.6B.100d.txt')

TEXT = Field(tokenize='spacy', tokenizer_language='en_core_web_sm', lower=True)

TEXT.build_vocab(train_dataset, vectors=GloVe(name='6B', dim=100))

weight_matrix = torch.zeros((len(TEXT.vocab), 100))

for word, index in vocab.stoi.items():
    weight_matrix[index] = embeddings.get(word, torch.randn(embedding_dim))

class EmbeddingLayer(nn.Module):

    def __init__(self, weights_matrix):
        super().__init__()
        num_embeddings, embedding_dim = weights_matrix.size()
        self.embeddings = nn.Embedding.from_pretrained(weights_matrix, freeze=False)

    def forward(self, input):
        return self.embeddings(input)


embedding_model = EmbeddingLayer(weight_matrix)

```

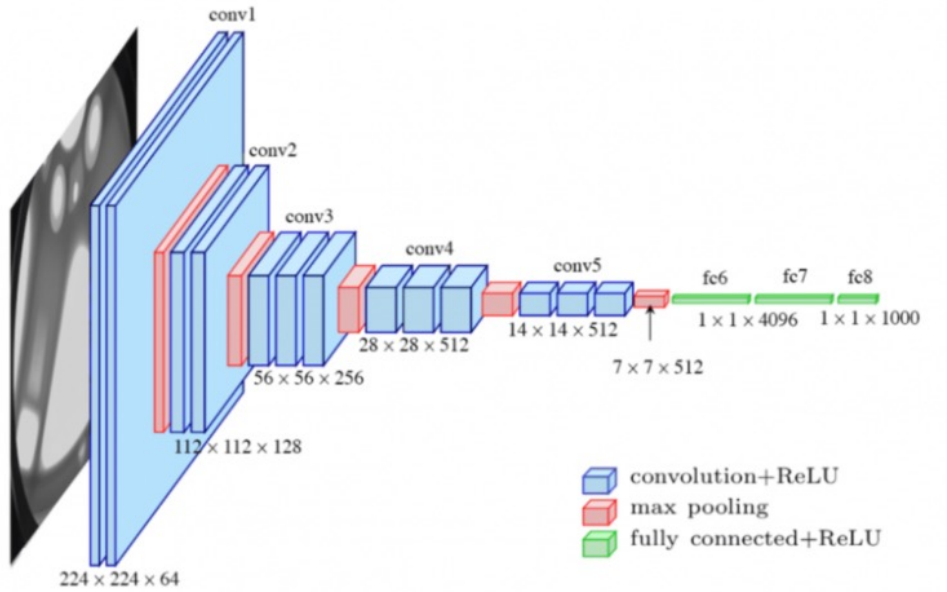
Network Geometries

The biggest remaining question is then:

How many layers, how wide, and what type?

As usual, “it depends,” but some general principles have emerged:

1. Deep networks are necessary for image processing, where the information is found in hierarchical groupings of image features:

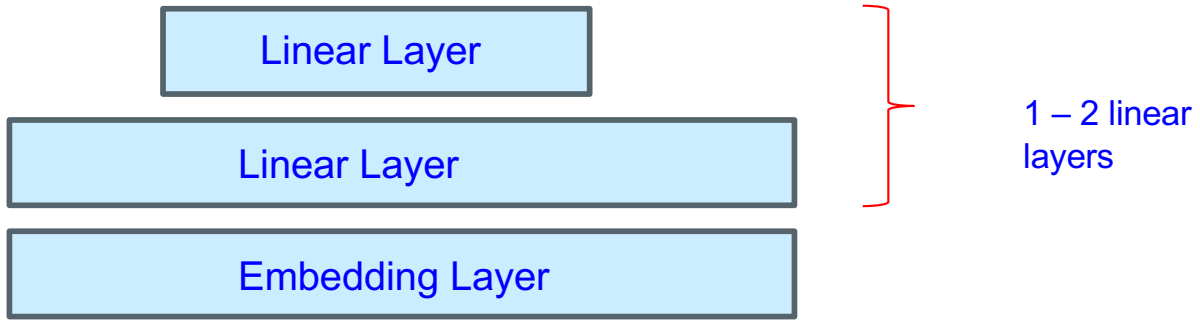


Network Geometries: How wide and deep?

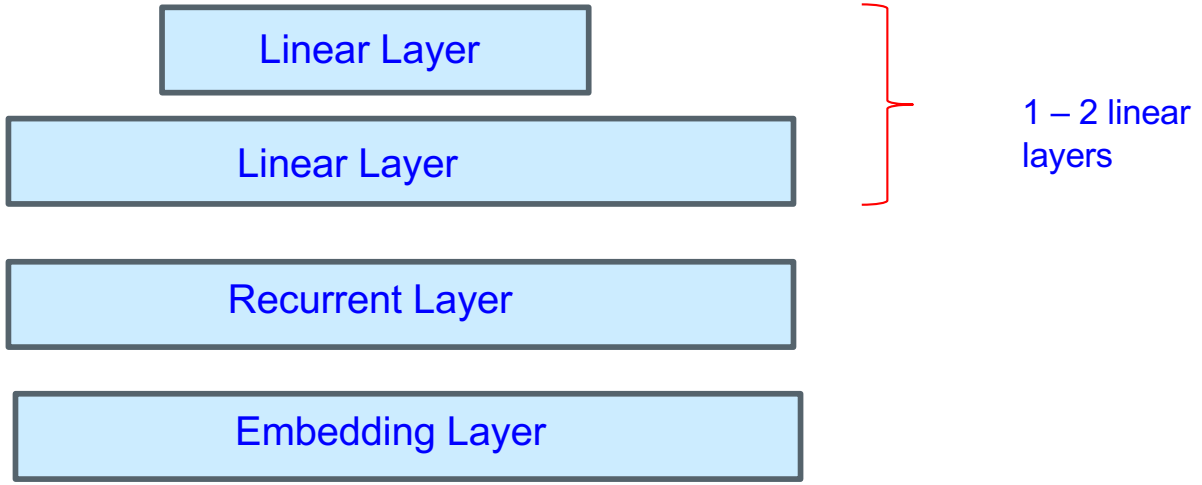
2. Information in text is sequential, with dependencies among words, but FFNNs and (B)RNNs have a limited ability to deal with long-range dependencies.

Hence, deep networks are not as useful – until we introduce transformers!

FFNN:

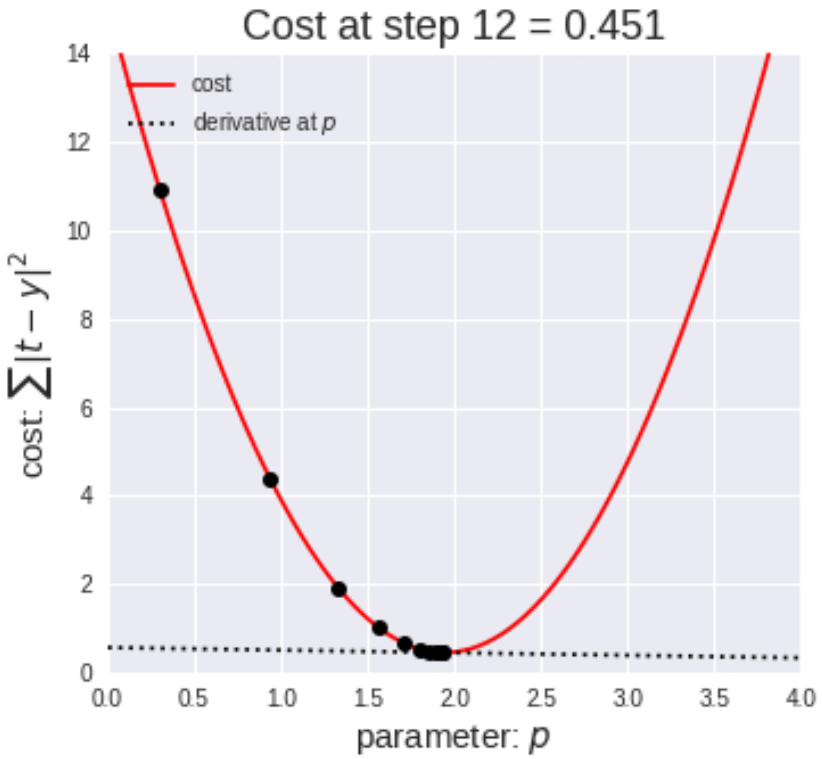


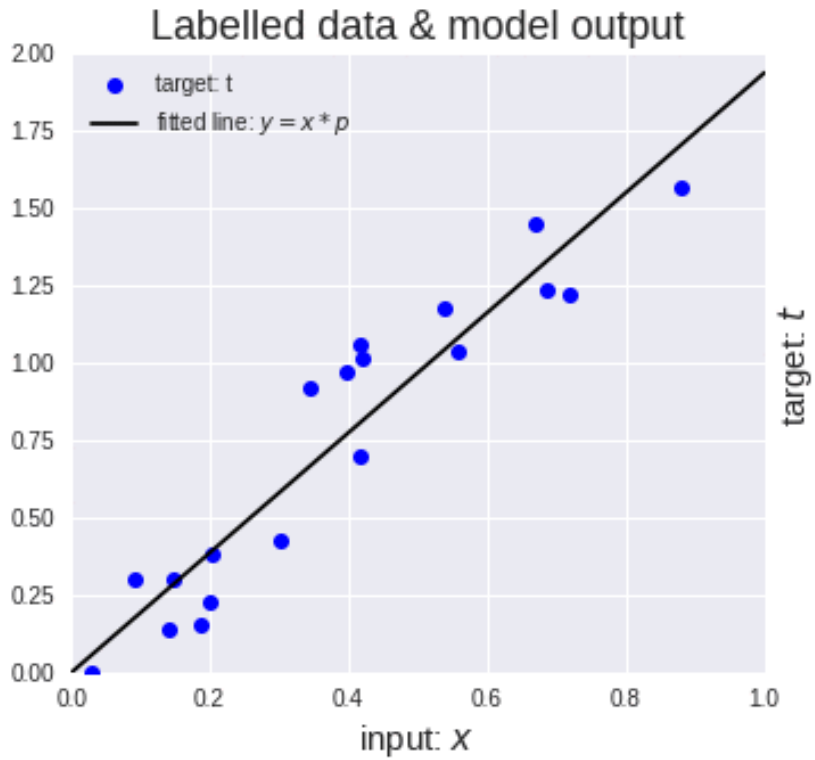
(B)RNN:



Network Geometries: How wide and deep?

3. Each neuron has essentially a linear discrimination power. You can (very roughly) consider each neuron having the ability to draw a line (or hyperplane) distinguishing two regions of your data.

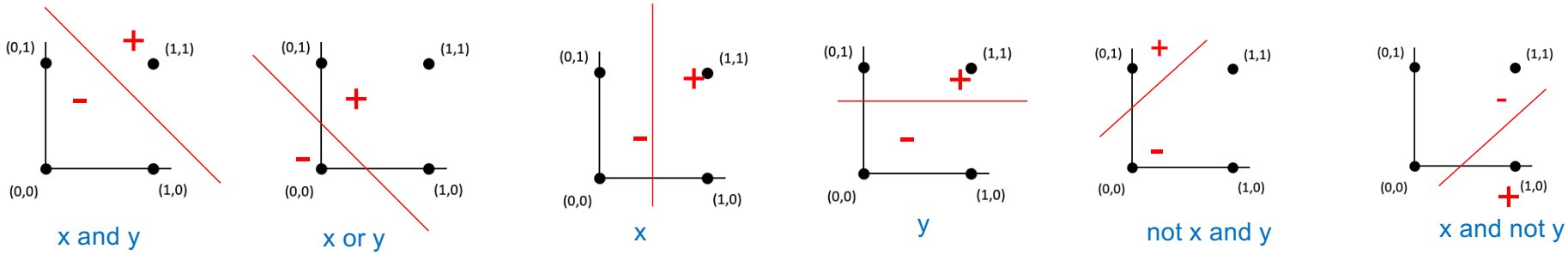




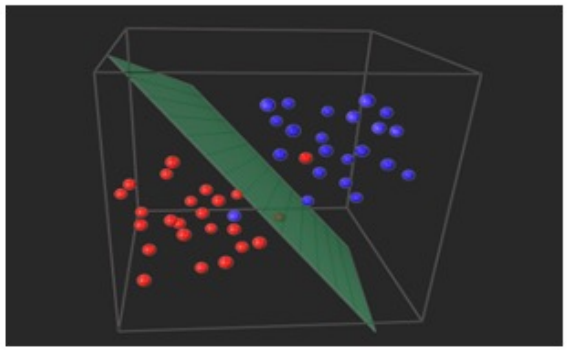
Network Geometries: How wide and deep?

Bumper Sticker Version: Each NN unit can draw a single line and hence can calculate Boolean functions of inputs.

Example: A single unit can perform binary classification on 4 points:

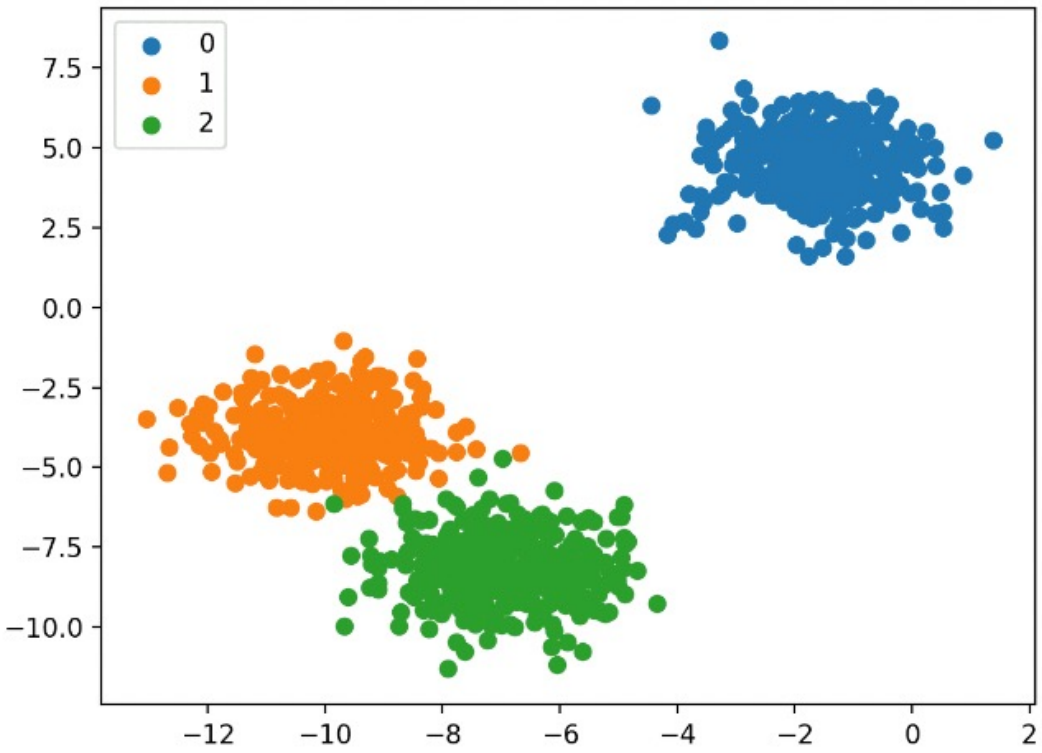


NOTE: This is happening in multiple dimensions, so we are really trying to draw hyperplanes!



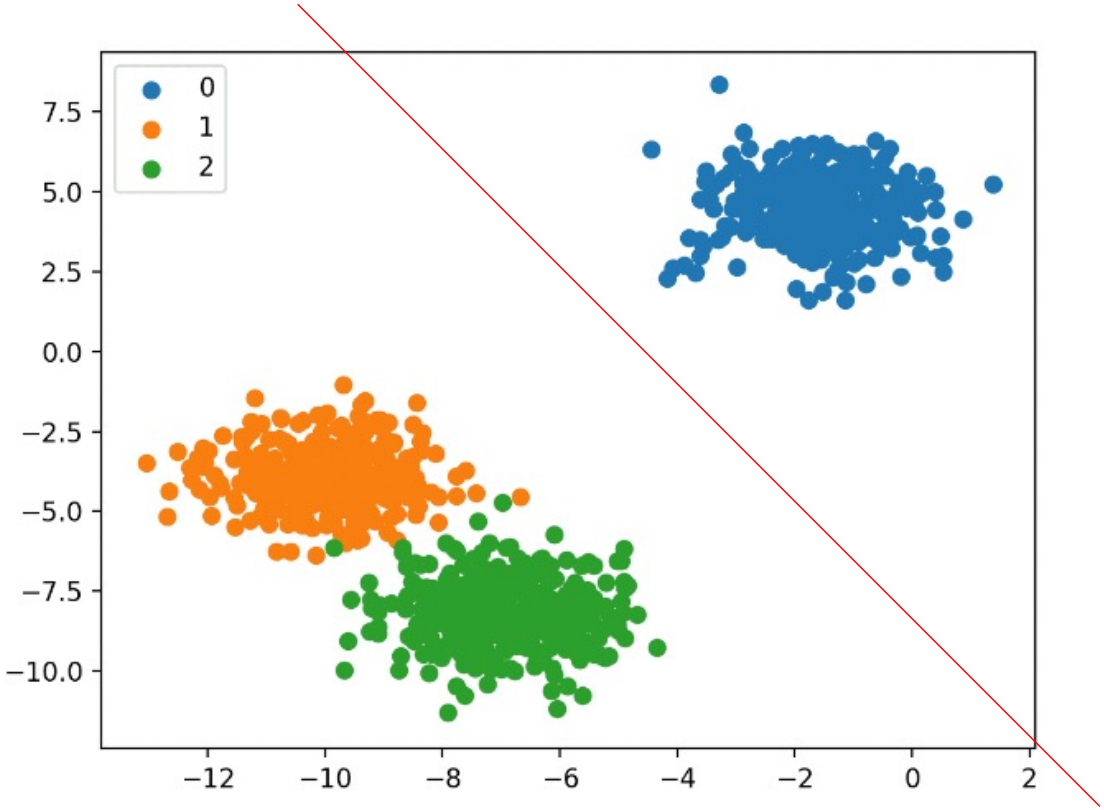
Network Geometries: How wide and deep?

Adding neurons allows us to draw multiple hyperplanes:



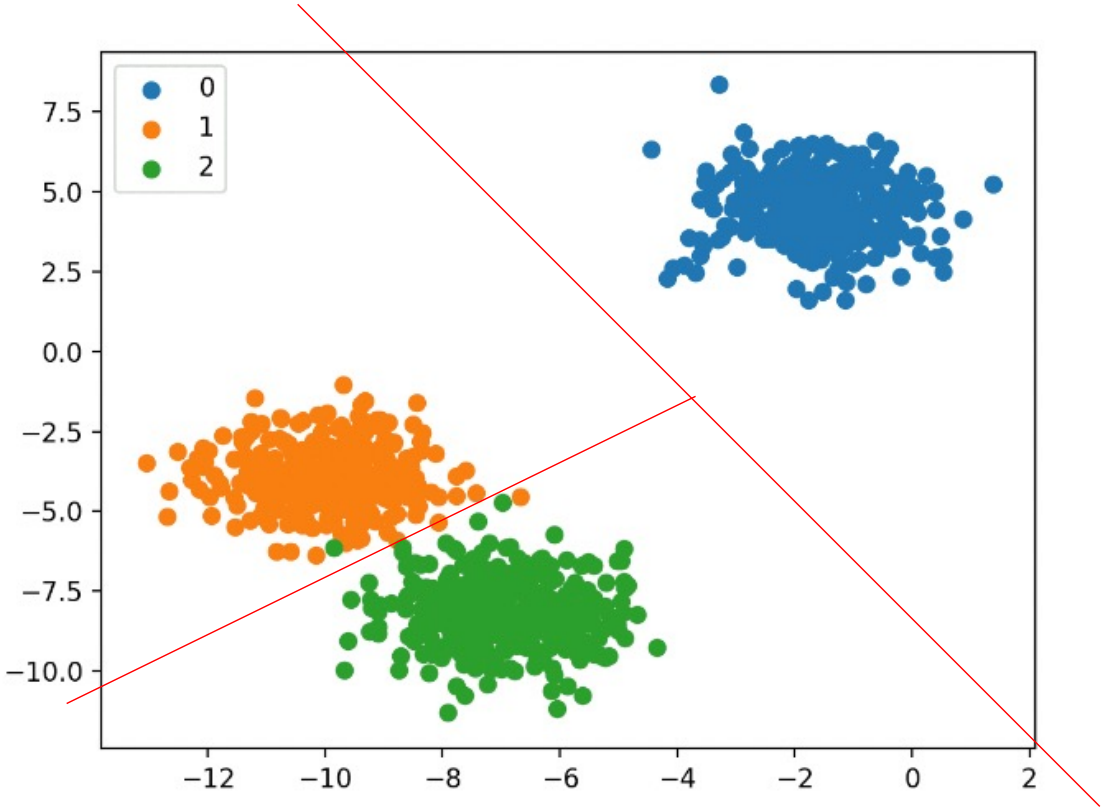
Network Geometries: How wide and deep?

Adding neurons allows us to draw multiple hyperplanes:



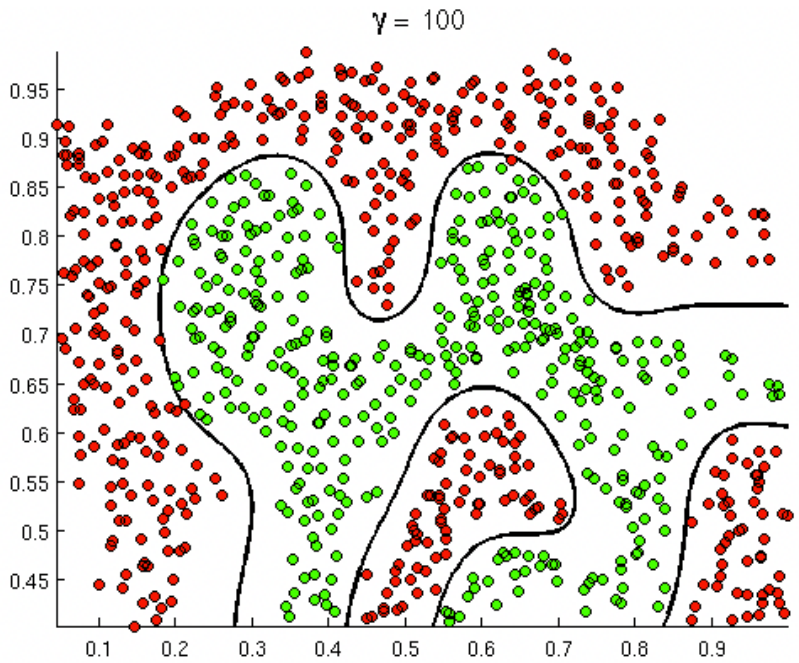
Network Geometries: How wide and deep?

Adding neurons allows us to draw multiple hyperplanes:



Network Geometries: How wide and deep?

Complex data sets involve lots of combinations of lines!



Network Geometries: How wide and deep?

Conclusions:

- For a given NLP data set, there is a particular width of layer that will have enough discriminatory power to do the task, and
- Adding more neurons will
  - Not necessarily improve its accuracy,
  - Lead to excessive training time,
  - More overfitting, and
  - Less ability to generalize.
- You are better off experimenting with different types of layers and **not** just increasing the depth and width.
- Recurrent networks are best for simpler NLP tasks (such as classification)
- Transformers are best for tasks involving sequence-to-sequence tasks such as machine translation, summarization, and conversation agents.

So.. On to transformers!