Deep Learning

Dhruv Madeka
Senior Machine Learning Scientist
Amazon
Courant Institute – Mathematics Seminar

All views are my own

What is learning?

- Traditional definition:
- Consider a task T, and a performance metric P
- An algorithm which improves it's performance P on a task T with experience E is said to learn from the experience E

What are the different tasks?

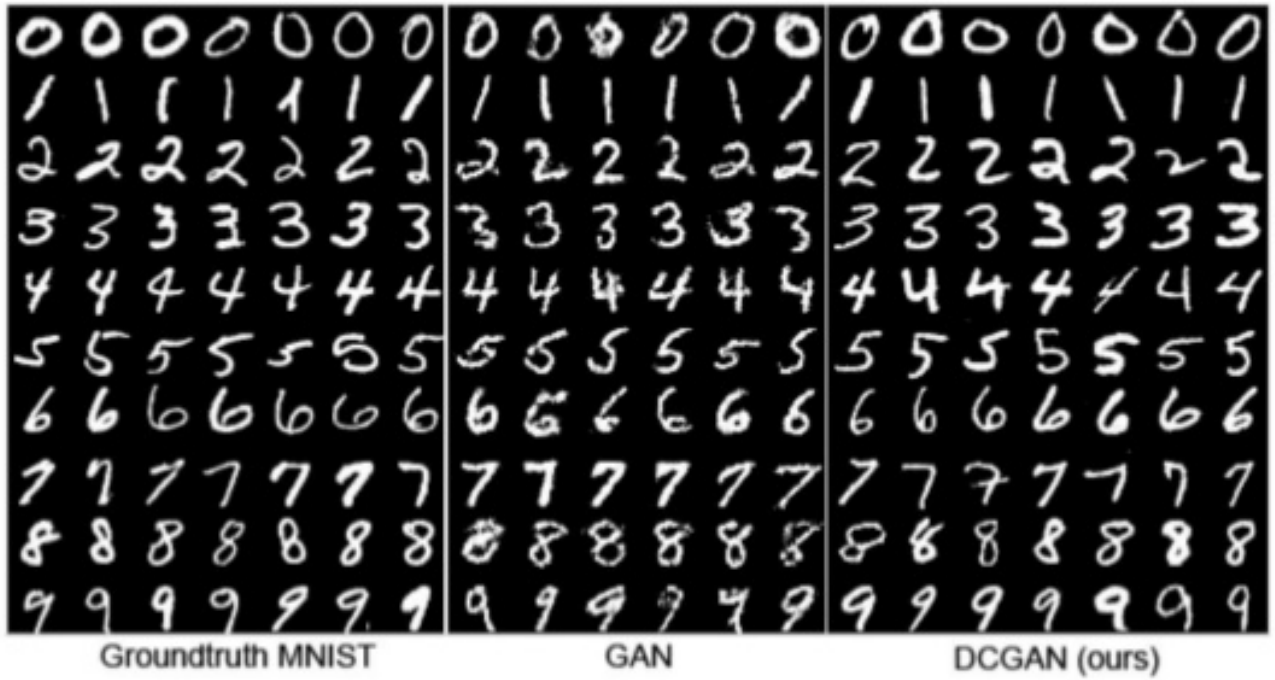
- Task can be:
  - Classification: $X \to \{1, \dots, k\}$
  - Regression: $X \to \mathbb{R}^{\mathbb{N}}$
  - Anomaly Detection
  - Etc. etc.
- More and more:
  - Denoising
  - Sampling
  - Machine Translation

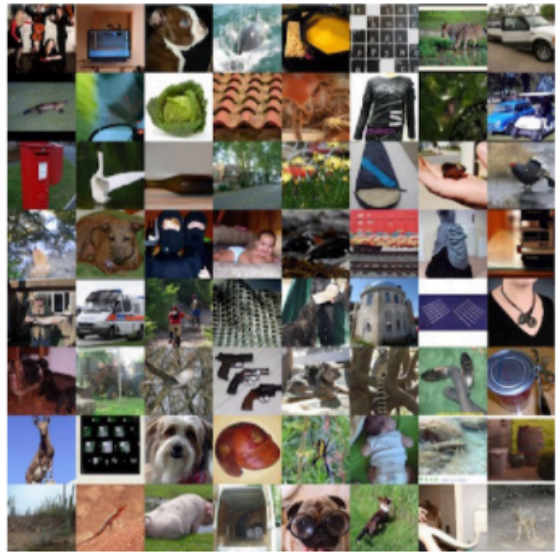
What are the different performance measures?

- Easier for classification/regression
- What about for unsupervised learning?
- How do we decide whether the samples we are generating are "good"?

What are the different performance measures?

- Are these GAN Images good?





What are the different performance measures?

- Does a higher BLEU score mean we're translating better?

- Criticisms:
  - Too much variation
  - Too little variation
  - Intrinsically meaningless
  - Human translators score low on BLEU

E: I am feeling hungry
H: मुझे भूख लग रही है
to-me hunger feeling is
I: मैं भूखा महसूस कर रहा हूँ
I hungry feel doing am

***n*-gram matches**: unigrams: 0/6; bi-grams: 0/5; trigrams: 0/4; 4-grams: 0/3

False Negative

False Positive

E: The Lok Sabha has 545 members
H: लोक सभा में ५४५ सदस्य हैं
Lok Sabha in 545 members are
I: लोक सभा के पास ५४५ सदस्य हैं
Lok Sabha has/near 545 members are

***n*-gram matches**: unigrams: 5/7; bi-grams:3/6; trigrams: 1/5; 4-grams: 0/4

Ananthakrishnan et al

Source: LeCun (2017)

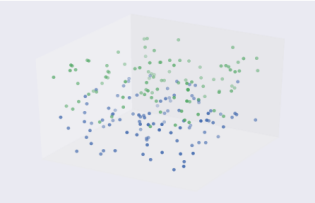
Why go deep?

Either the world is compositional or there is a god – Eisner, ICLR (2014)

Traditional P Modelling

Feature Extraction → Classifier/ Regressor

Modern Learning

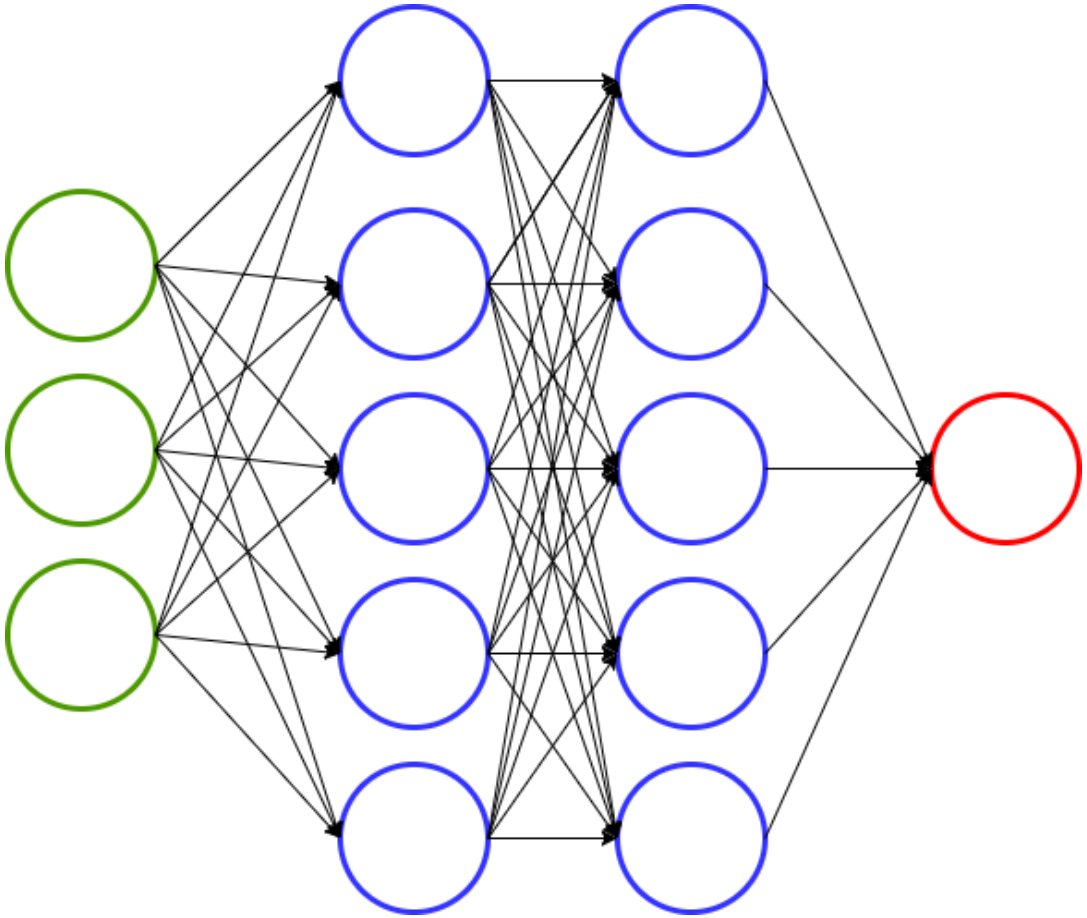


Feature Extraction → Mid-level features → Classifier/ Regressor

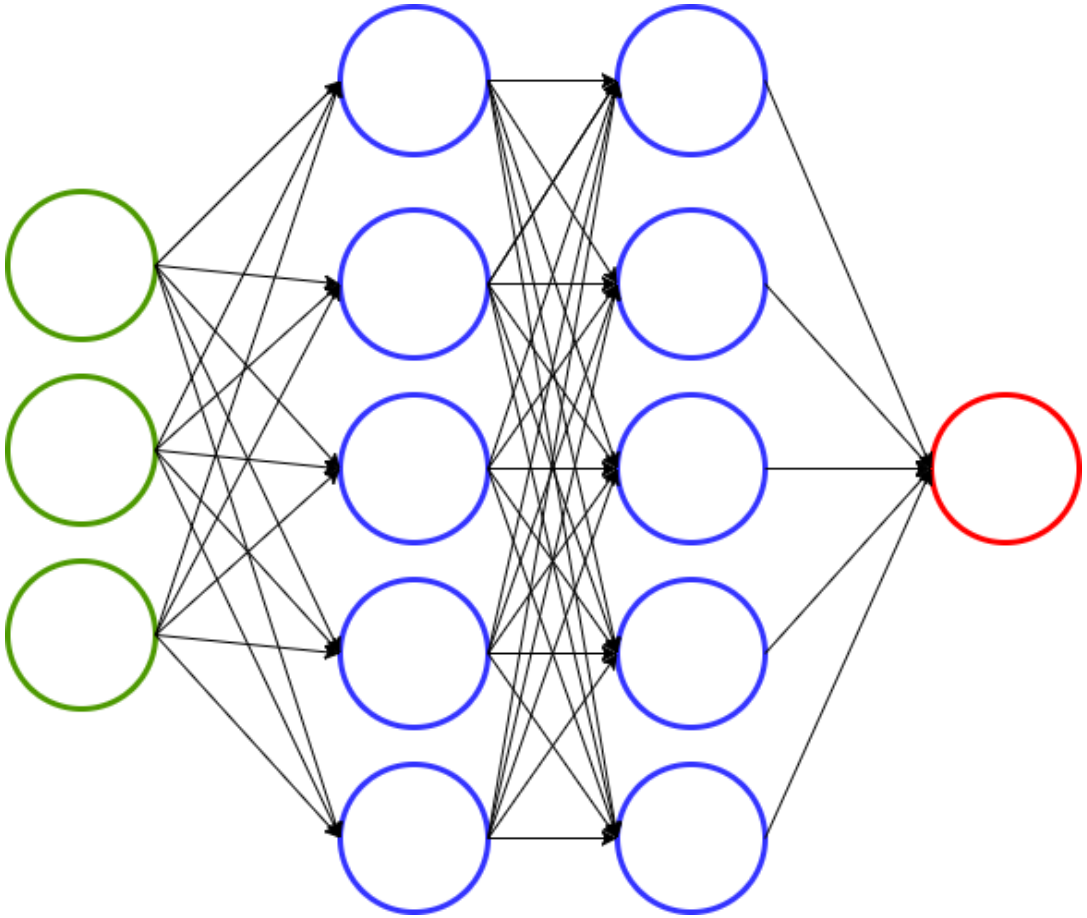
Going Deep

Low Level Features → Mid-level features → High level features → Classifier/ Regressor

So, what are those funny little diagrams?



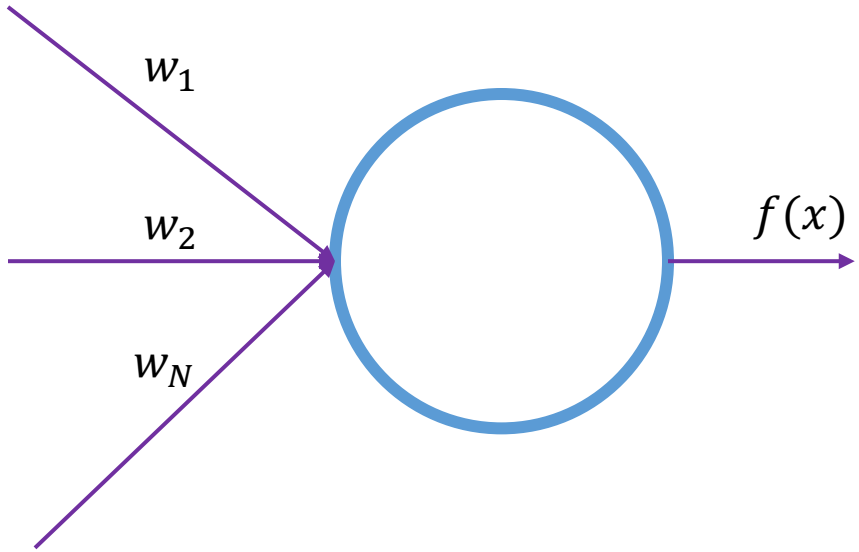
So, what are those funny little diagrams?



Each node is computes a weighted sum of it's inputs and passes it through an element wise non-linearity

Multiple layers of inputs creates a compositional hierarchy

So, what are those funny little diagrams?



Each node is computes a weighted sum of it's inputs and passes it through an element wise non-linearity

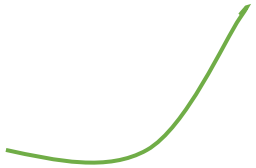
$$f(x) = \sigma\left(\sum_i w_i x_i\right)$$

ReLU (Call payoff):

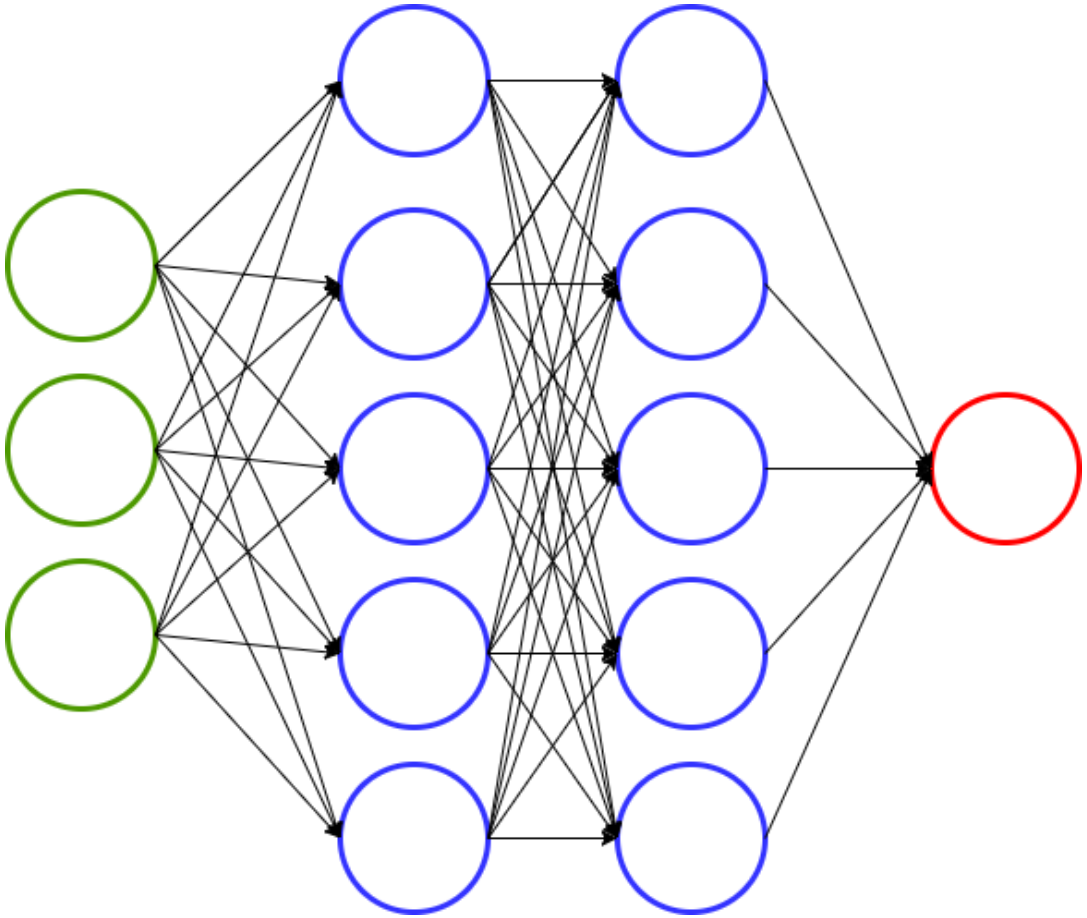
Maxout $\max(x_1, \dots, x_N)$

Sigmoid/Tanh (Saturating):

SeLU



So, what are those funny little diagrams?



Each node is computes a weighted sum of it's inputs and passes it through an element wise non-linearity

$$\sigma\left(\sum_i w_i x_i\right)$$

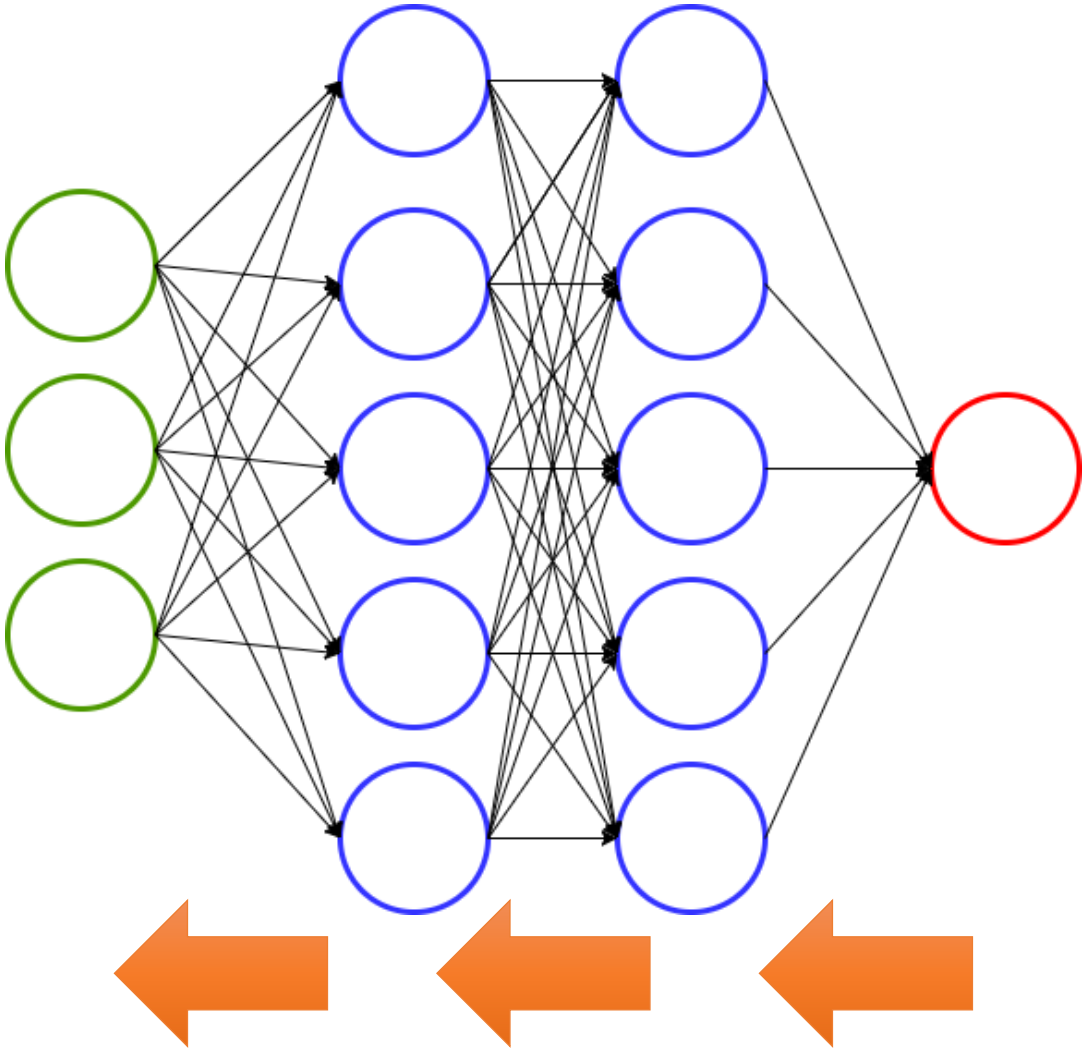
Multiple layers of inputs creates a compositional hierarchy

$$\sigma\left(\sum_j w_j^2 \sigma\left(\sum_i w_i^1 x_i\right)\right)$$

Universal Approximation Theorem

- Shallow and Wide vs Deep and Narrow – More memory or more time
- Universal Approximation Theorem states that a network with a single hidden layer can approximate any continuous function
- Not surprising – when it’s wide enough, it’s basically a lookup table
- But most interesting functions require a very large lookup table – depth helps us reduce the number of neurons exponentially!

That sounds fun to train:



Backprop (Rumelhart, 1986) is maybe the reason the whole thing works

Before we go into that, let's talk about what gradient descent is first

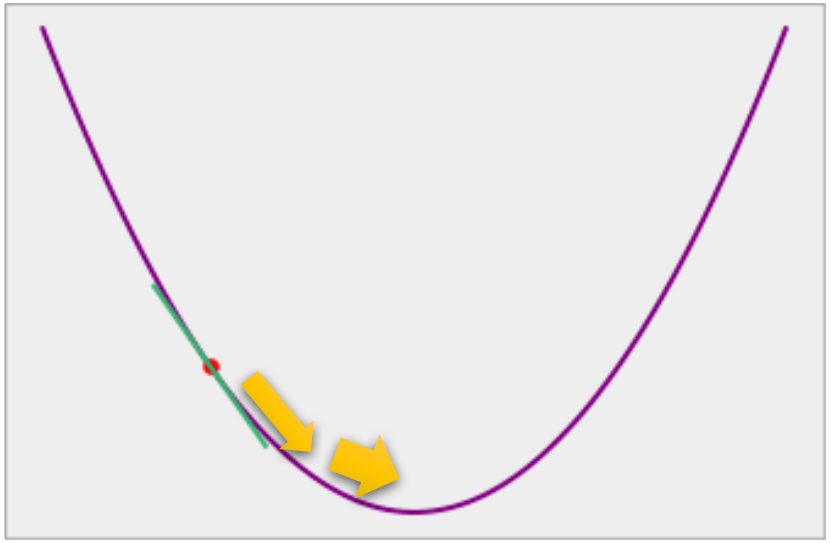
Gradient Descent

$$D_u f(x) = \nabla f(x).u = ||\nabla f(x)|| cos\theta$$

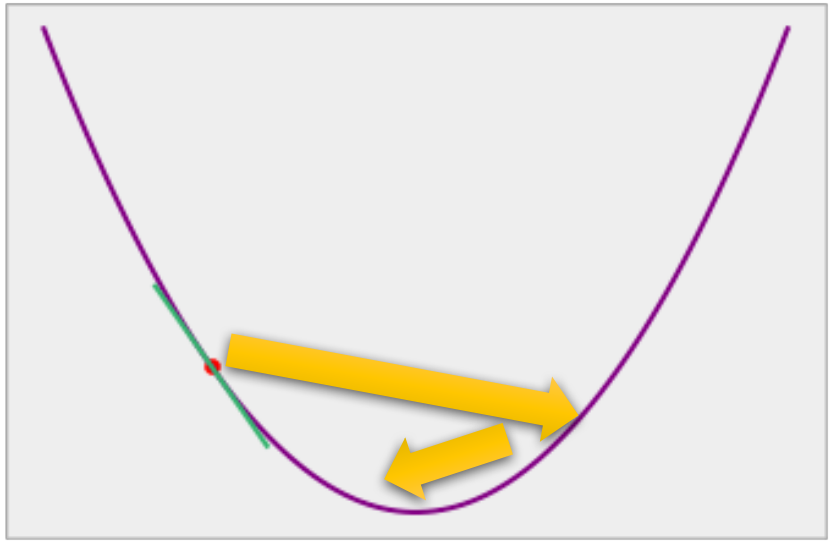
$\Longrightarrow D_u f(x)$ is maximized in the direction of $\nabla f(x)$

$$w_{n+1} = w_n - \eta \nabla f(w)$$

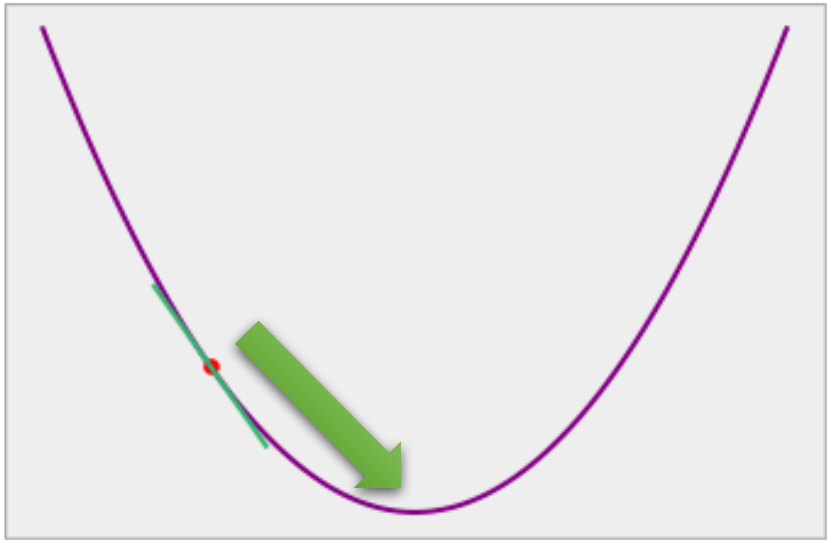
Gradient Descent



$\eta < \eta_{opt}$



$\eta > \eta_{opt}$



$\eta = \eta_{opt}$

$$L(w) = L(w_0) + \frac{1}{2}a(w - w_0)^2$$

$$\Rightarrow \frac{\partial \mathrm{L(w)}}{\partial w} = \mathrm{a(w - w_0)}$$

$$\Rightarrow w = w_0 + a^{-1}\frac{\partial L(w)}{\partial w}$$

$$\Rightarrow \frac{\partial^2 L(w)}{\partial w^2} = a$$

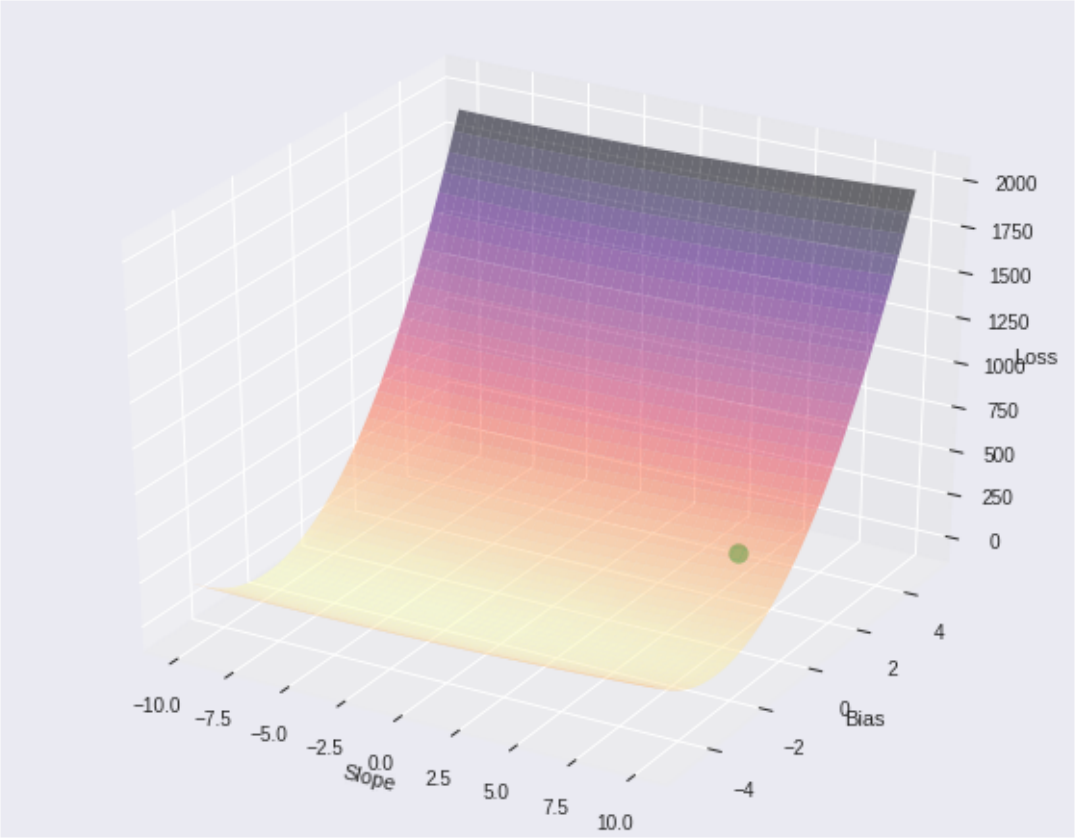
$$\eta_{opt} = \left(\frac{\partial^2 L(w)}{\partial w^2}\right)^{-1}$$

Gradient Descent

- For linear regression, the loss function usually looks like

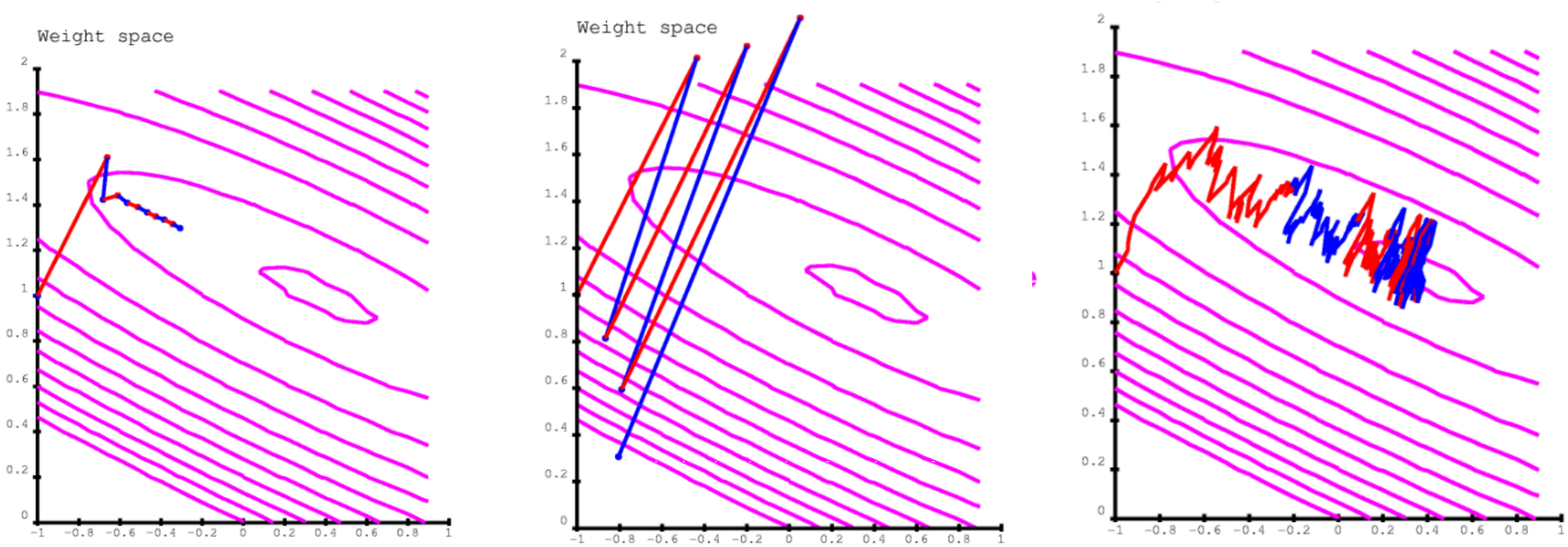
  - $L(w) = \frac{\sum_i (y_i - (w_0 + w_1 x_i))^2}{m}$
  - $\nabla L(w) = \begin{pmatrix} \frac{\sum_i (y_i - (w_0 + w_1 x_i))}{m} \\ \frac{\sum_i \left(y_i - (w_0 + w_1 x_i)\right) \cdot x_i}{m} \end{pmatrix}$

Pretty expensive if m is large. Do we really need all those points?



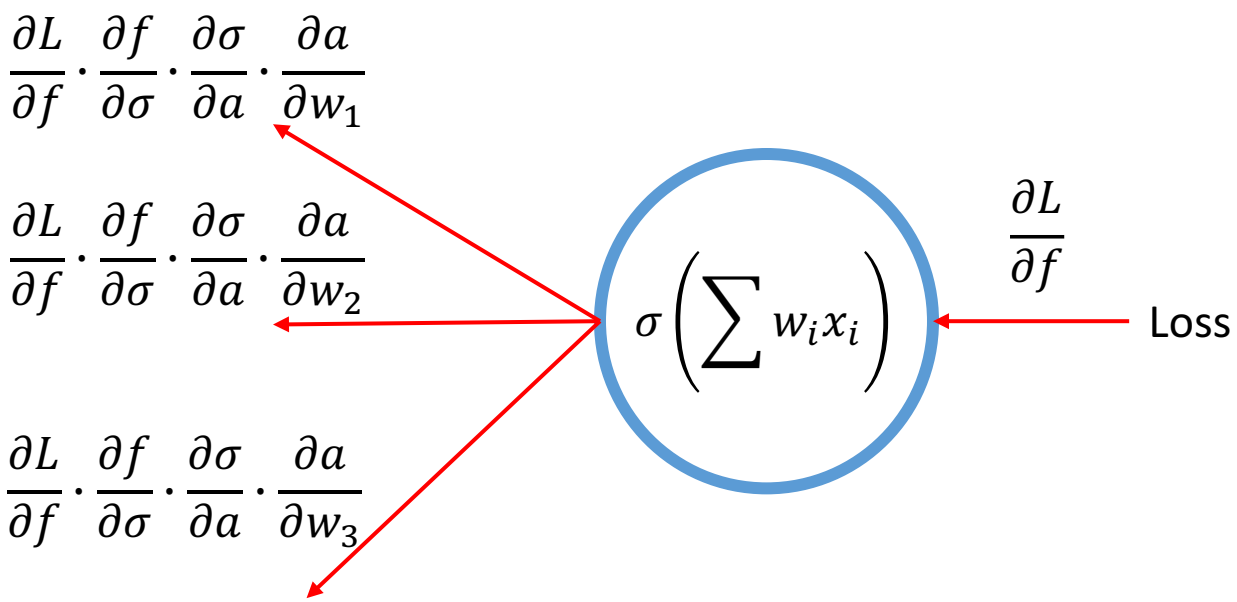
Stochastic Gradient Descent

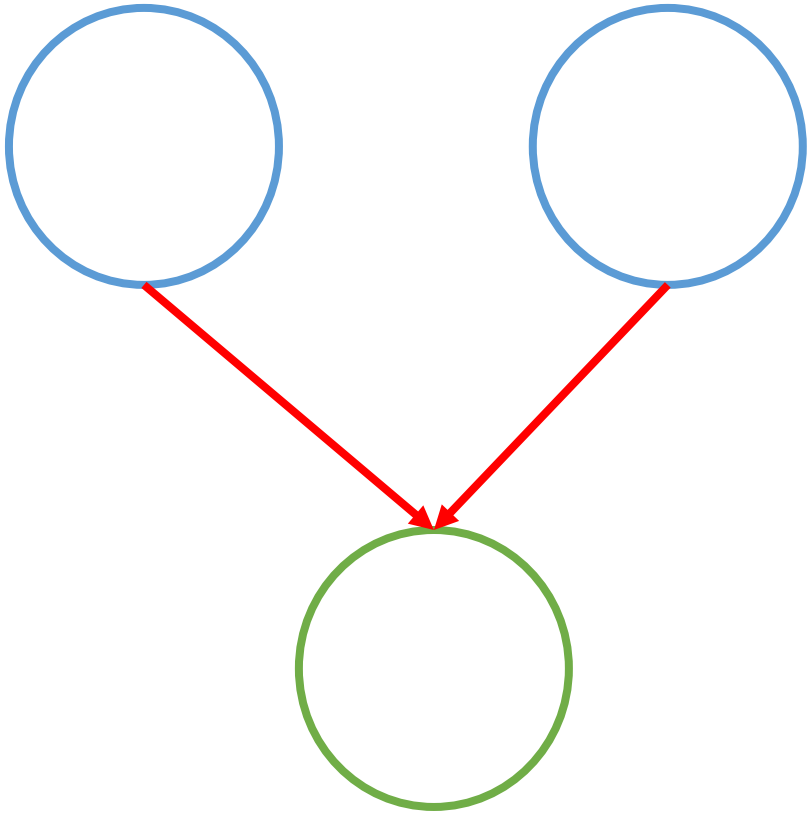
- Consists of showing the input vector for a few examples
- Compute loss
- Use average gradient as a noisy estimate of the true gradient



Source: [LeCun (2017)]

Backpropagation





$$\frac{\partial L}{\partial X_i} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial x_{i+1}} \cdot \frac{\partial x_{i+1}}{\partial x_i}$$

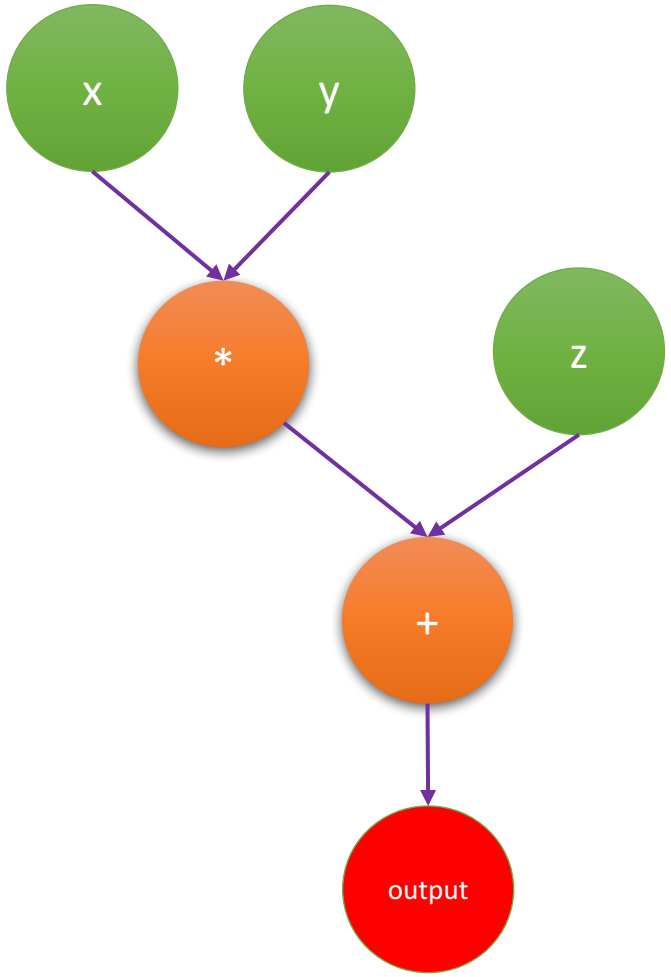
$$\frac{\partial L}{\partial W_i} = \frac{\partial L}{\partial X_i} \cdot \frac{\partial \sigma(x_{i-1}, w_i)}{\partial w_i}$$

This sounds hard to implement

```python
import numpy as np

x = np.random.randn(3, 3)
y = np.random.randn(3, 3)
z = np.random.randn(3, 3)

a = x * y
output = a + z
```




```python
import mxnet as mx

x = mx.nd.random_normal(shape=(3, 3))
y = mx.nd.random_normal(shape=(3, 3))
z = mx.nd.random_normal(shape=(3, 3))

a = x * y
output = a + z
```

With numpy, sure – with MXNet, not so much

```python
import numpy as np

x = np.random.randn(3, 3)
y = np.random.randn(3, 3)
z = np.random.randn(3, 3)

a = x * y
b = a + z
c = 2 * b
output = 3 * c

grad_outputc = 3 * np.ones((3, 3))
grad_bc = 2 * np.ones((3, 3))
grad_ba = 1 * np.ones((3, 3))
grad_ax = y
grad_outputx = grad_outputc * grad_bc * grad_ba * grad_ax
print(grad_outputx)
```

```python
import mxnet as mx

x = mx.nd.random_normal(shape=(3, 3))
y = mx.nd.random_normal(shape=(3, 3))
z = mx.nd.random_normal(shape=(3, 3))

x.attach_grad()

with mx.autograd.record():
    a = x * y
    b = a + z
    c = 2 * b
    output = 3 * c

output.backward()
print(x.grad)
```

Training a simple model - manually

Create variables

Record forward pass

Update parameters

```
epochs = 10
learning_rate = .0001
num_batches = num_examples/batch_size

w = nd.random_normal(shape=(num_inputs, num_outputs), ctx=model_ctx)
b = nd.random_normal(shape=num_outputs, ctx=model_ctx)
params = [w, b]

def net(X):
    return mx.nd.dot(X, w) + b

for e in range(epochs):
    cumulative_loss = 0
    # inner loop
    for i, (data, label) in enumerate(train_data):
        data = data.as_in_context(model_ctx)
        label = label.as_in_context(model_ctx).reshape((-1, 1))
        with autograd.record():
            output = net(data)
            loss = mx.nd.mean((output - label) ** 2)
        loss.backward()
        for param in params:
            param[:] = param - lr * param.grad
        cumulative_loss += loss.asscalar()
    print(cumulative_loss / num_batches)
```

nn Module – Life is much simpler

Essentially, replace simple dot product with this

```
num_hidden = 64
net = gluon.nn.Sequential()
with net.name_scope():
    net.add(gluon.nn.Dense(num_hidden, activation="relu"))
    net.add(gluon.nn.Dense(num_hidden, activation="relu"))
    net.add(gluon.nn.Dense(num_outputs))
```

Replace optimization step with advanced optimizer

```
trainer = gluon.Trainer(net.collect_params(), 'adam', {'learning_rate': .01, 'beta': 0.99})
```

```
with autograd.record():
    output = net(data)
    loss = softmax_cross_entropy(output, label)
loss.backward()
trainer.step(data.shape[0])
```

Blocks allow simplified abstractions for complicated architectures

Define Layers

Define forward pass

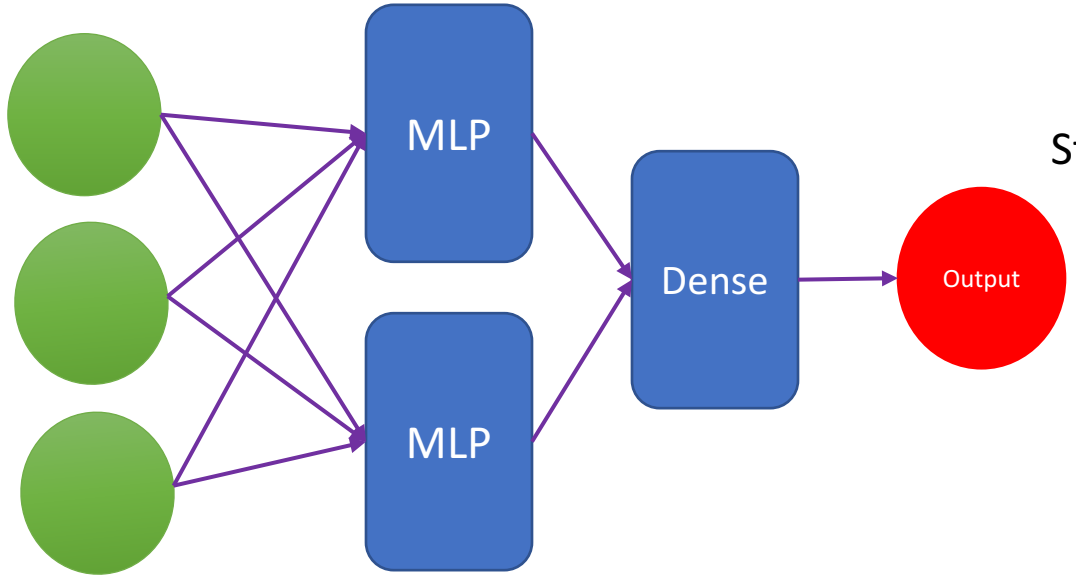
```
class MLP(gluon.Block):
    def __init__(self, **kwargs):
        super(MLP, self).__init__(**kwargs)
        with self.name_scope():
            self.dense0 = gluon.nn.Dense(64)
            self.dense1 = gluon.nn.Dense(64)
            self.dense2 = gluon.nn.Dense(10)

    def forward(self, x):
        x = nd.relu(self.dense0(x))
        x = nd.relu(self.dense1(x))
        x = self.dense2(x)
        return x
```

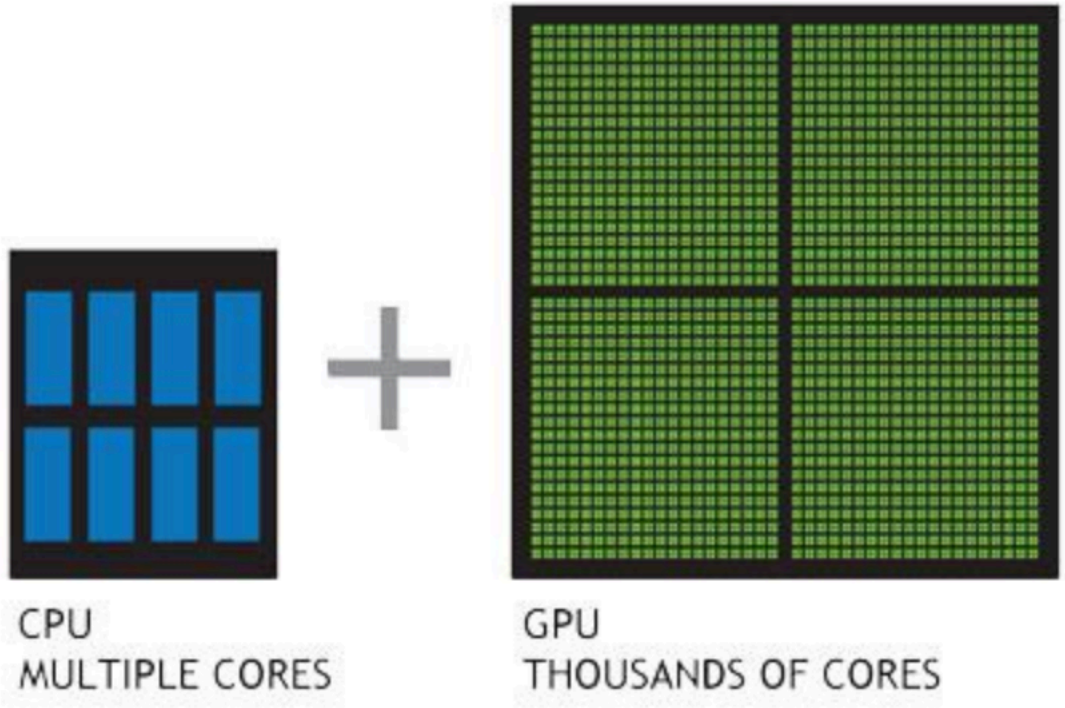
Stacked Architecture

```
class MyBlock(gluon.Block):
    def __init__(self, **kwargs):
        super(MyBlock, self).__init__(**kwargs)
        with self.name_scope():
            self.mlp1 = MLP()
            self.mlp2 = MLP()
            self.dense = gluon.nn.Dense(5)

    def forward(self, x):
        x1 = self.mlp1(x)
        x2 = self.mlp2(x)
        x = mx.nd.concat(x1, x2)
        return mx.nd.relu(self.dense(x))
```



CPUs vs GPUs



CPUs are optimized for latency
GPUs for bandwidth

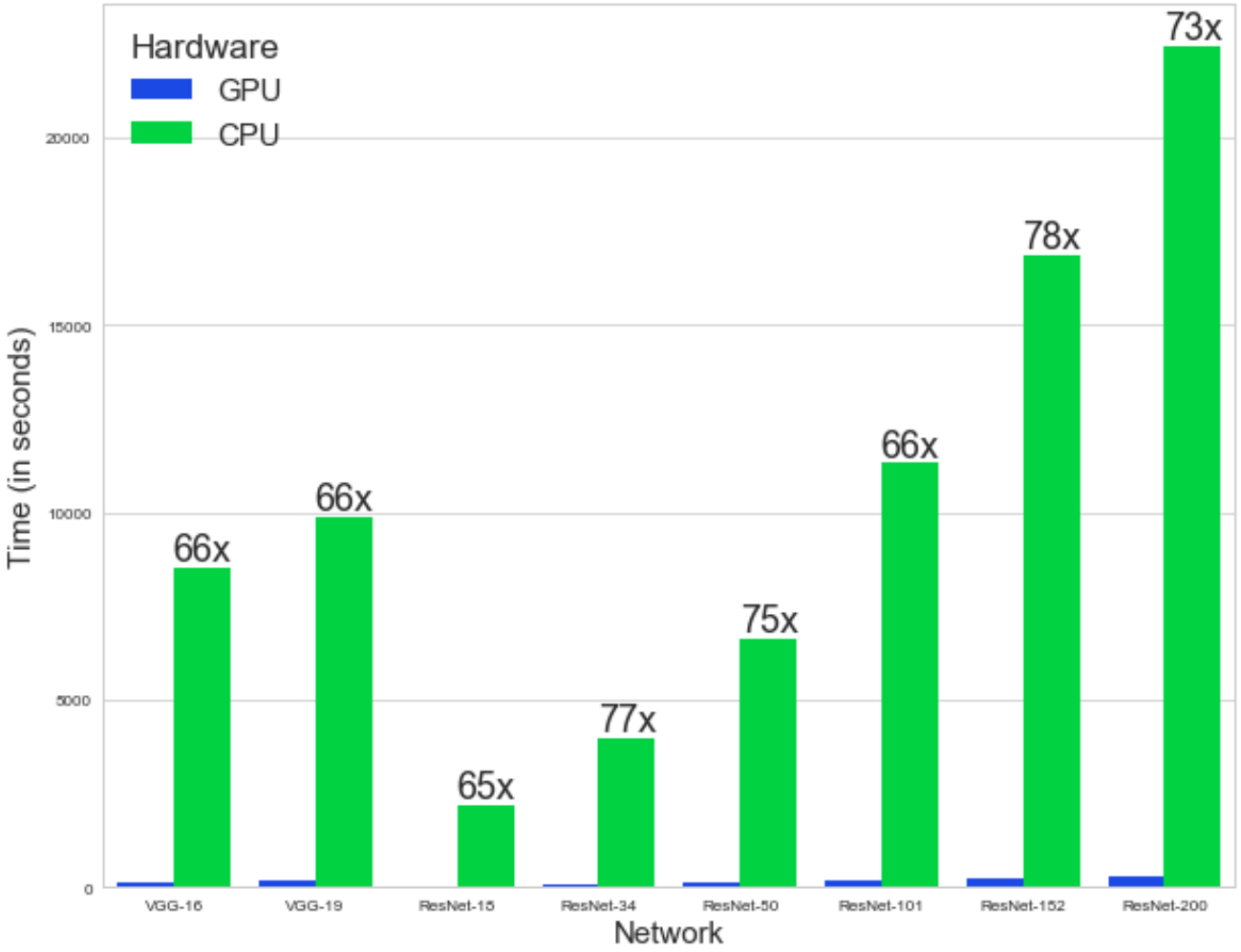
CPU can fetch small amount of memory really quickly – GPU can fetch large amounts of memory in the same time

The large number of cores in the GPU hide the latency issues by parallelizing the operations over a lot of threads

This is important for Deep Learning – since most of it is large matrix multiplications

Source: NVidia

CPUs vs GPUs



GPU - GTX 1080 Ti:

- 3584 Cuda Cores
- 10.6 FP32 TFLOPS
- 11GB Memory

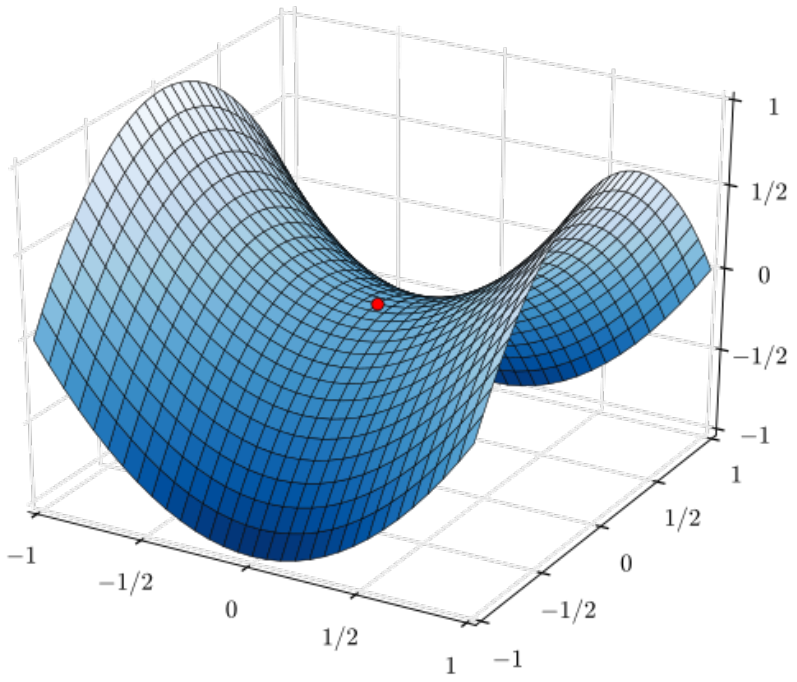
Intel® Xeon® Processor E5-2630 v3:

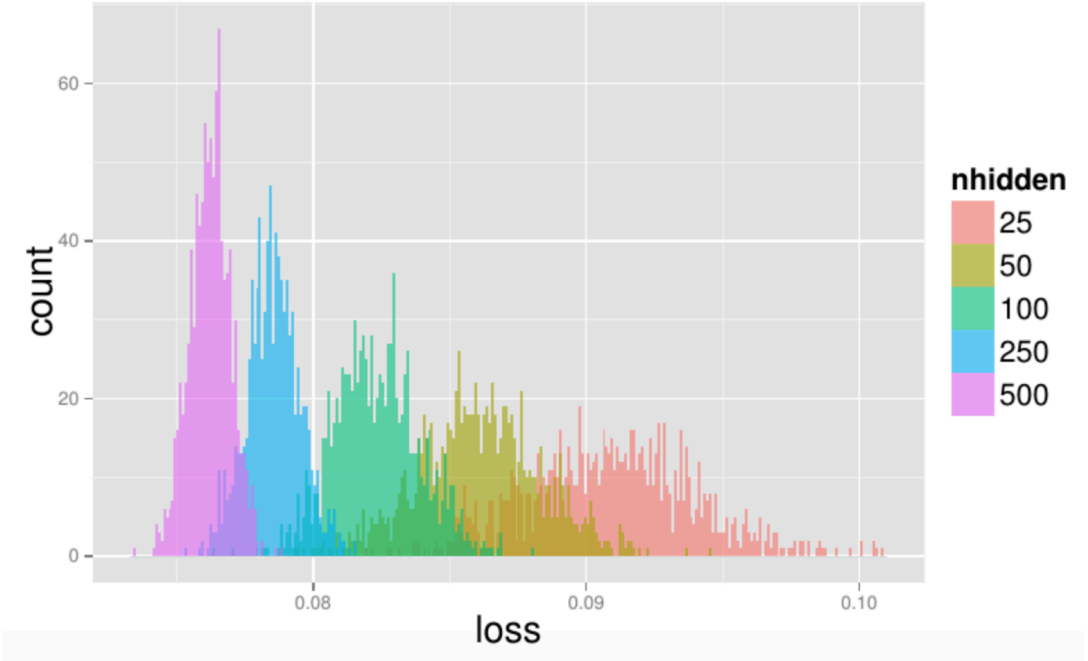
- 8 Cores
- 20MB Cache

Source: [jcjohnson (Github)]

Some Theory

- A lot of what you hear as criticism of Deep Learning typically used to involve (and still does) that you don't have any guarantee of converging to a minimum
- Ben Arous et al [2014] show that this doesn't really matter
- Using higher order spherical spin-glass based results, they show that any deep rectified network has an enormous number of critical points
- Seems to show that the number of saddle points with fewer downward curving points are very large in number but all have the same value!





Regularization in Deep Learning

Different types of "conventional" regularization

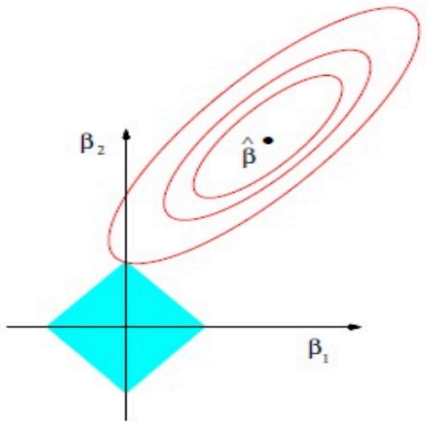
L1 Regularization

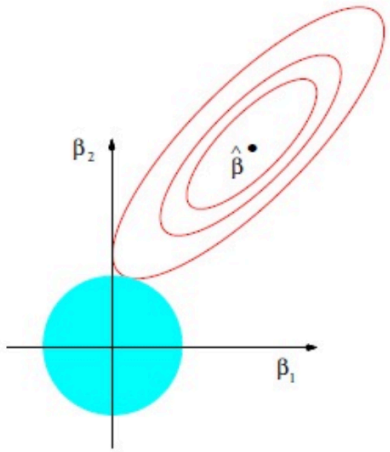
- Quite similar to conventional modeling – penalize the L1 Norm of the weights
- Cortes et al [2016]
- $R(f) \leq \widehat{R_{s,\rho}} + \frac{4}{\rho}\sum_{k=1}^{l}||w_k||_1 \mathbb{R}_m(\mathbb{H}_k) + K(\rho, m, l, d)$
- Seems to say penalize higher layers more than lower layers
- If you're overfitting – typically it works better to have a wider first layer and penalize it heavily and thinner higher layers

L2 Regularization

- Penalizes the L2-Norm of the weights
- Typically implemented as weight decay in many libraries
- Rewrites the objective as: $L'(X, \Omega, w) = L(X, \Omega, w) + \frac{\lambda}{2}||w||^2$
- Taking derivatives: $\nabla L'(X, \Omega, w) = \nabla \mathrm{L} + \lambda w$

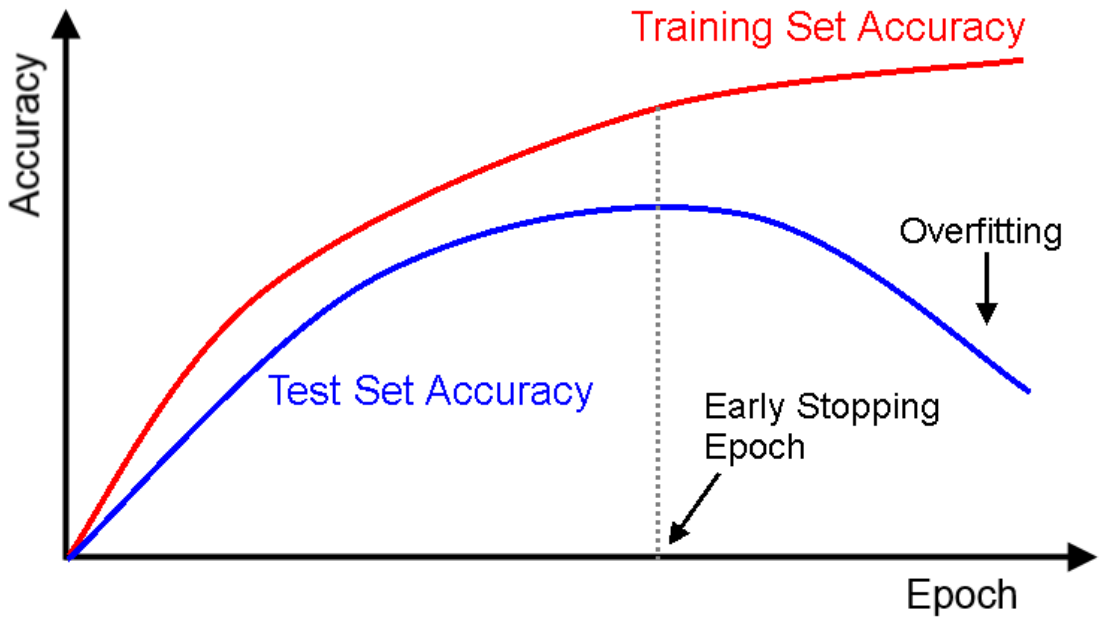
$$w_{n+1} = w_n - \eta \nabla L'$$
$$w_{n+1} = w_n - \eta \nabla L - \eta \lambda w_n$$
$$w_{n+1} = (1 - \eta\lambda) w_n - \eta \nabla L$$





More interesting forms of regularization

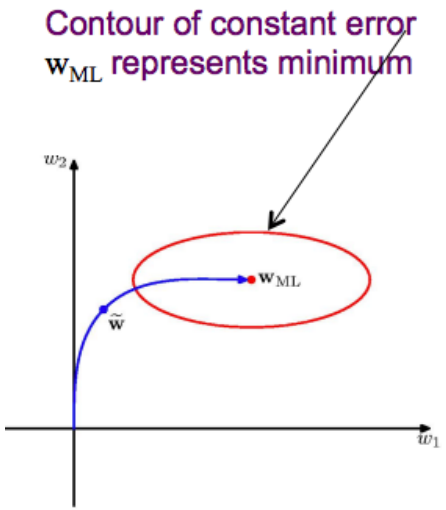
- Implicit regularization:
  - Early Stopping: During training, monitor the error on a validation set and stop when that error meets some criteria



$$w_{n^*+1} = w_{n^*} - \lambda\eta w_{n^*} + \eta\nabla L$$

$$\Rightarrow \lambda \sim \frac{||\nabla L||}{||w||}$$

$\frac{1}{||w||} \rightarrow$ not a bad approximation



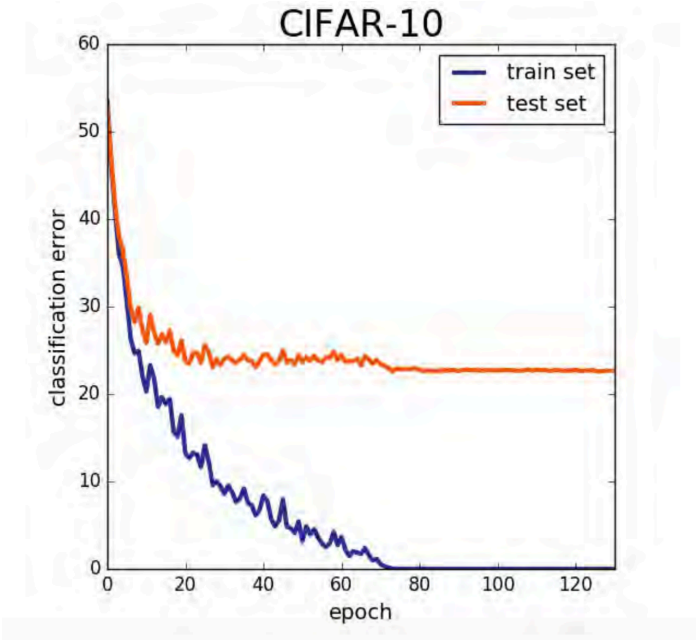
More interesting forms of regularization

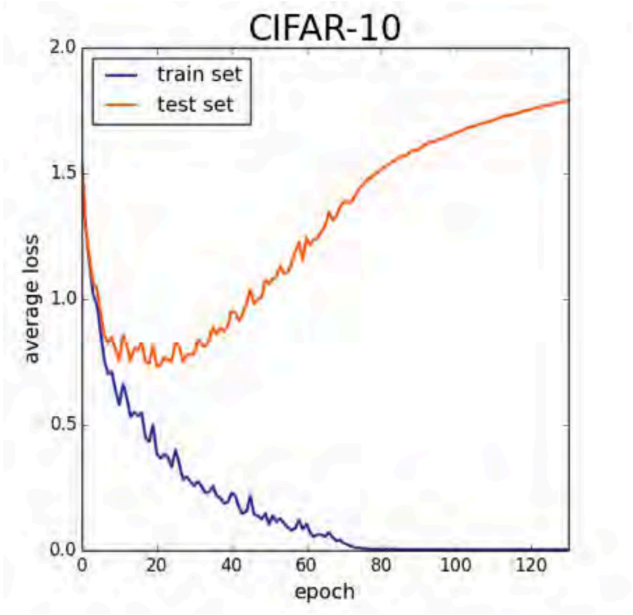
- Implicit regularization:
  - Stochastic Gradient Descent: Recht et al [2015] use linear regression to hypothesize that the gradient descent finds good “margin” solutions

$$\min_{w} \ (w^T x - y)^2$$

$$SGD \rightarrow \min_{s.t.\ Xw = y} ||w||$$

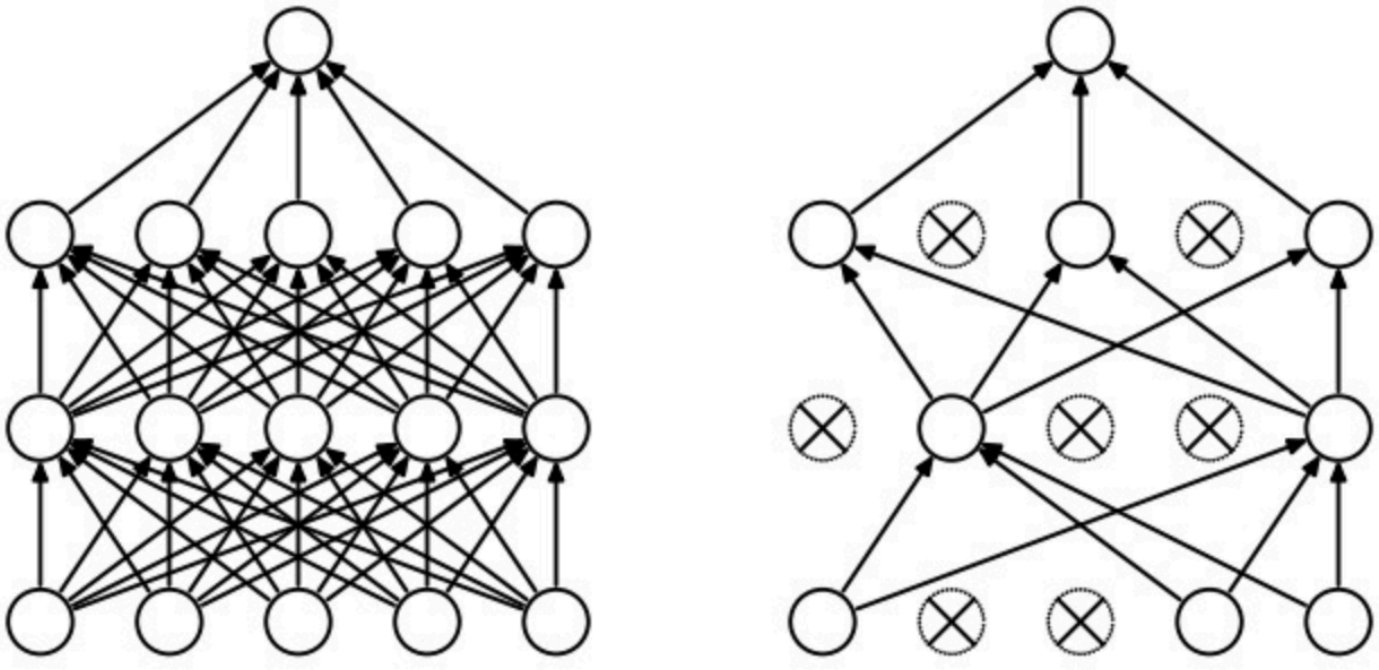
$\frac{1}{||w||}$ is the margin of the classifier





Workhorses

- Dropout:



(a) Standard Neural Net (b) After applying dropout.

For every single neuron (during training) – replace the output by

$$f(h) = D \odot a(h)$$

Where D is a vector of Bernoulli i.i.d's with probability p of being zero

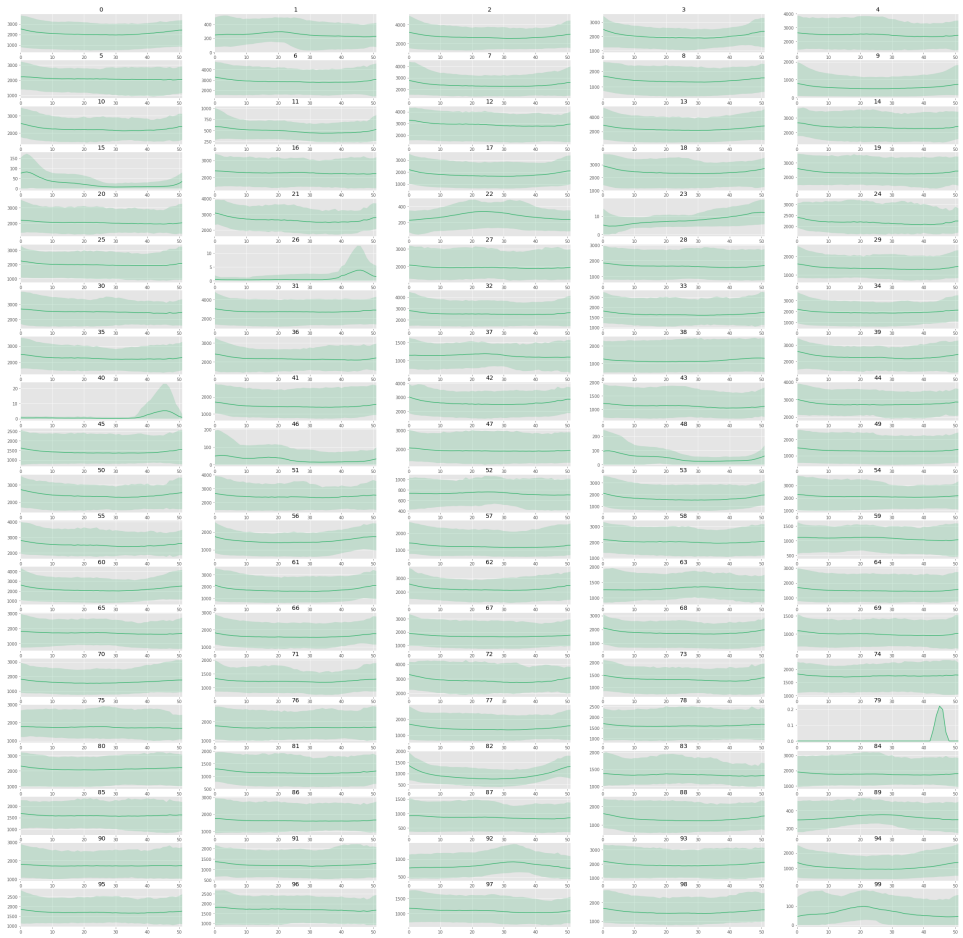
Masking effectively trains an ensemble of *weak learners* on different mini-batches

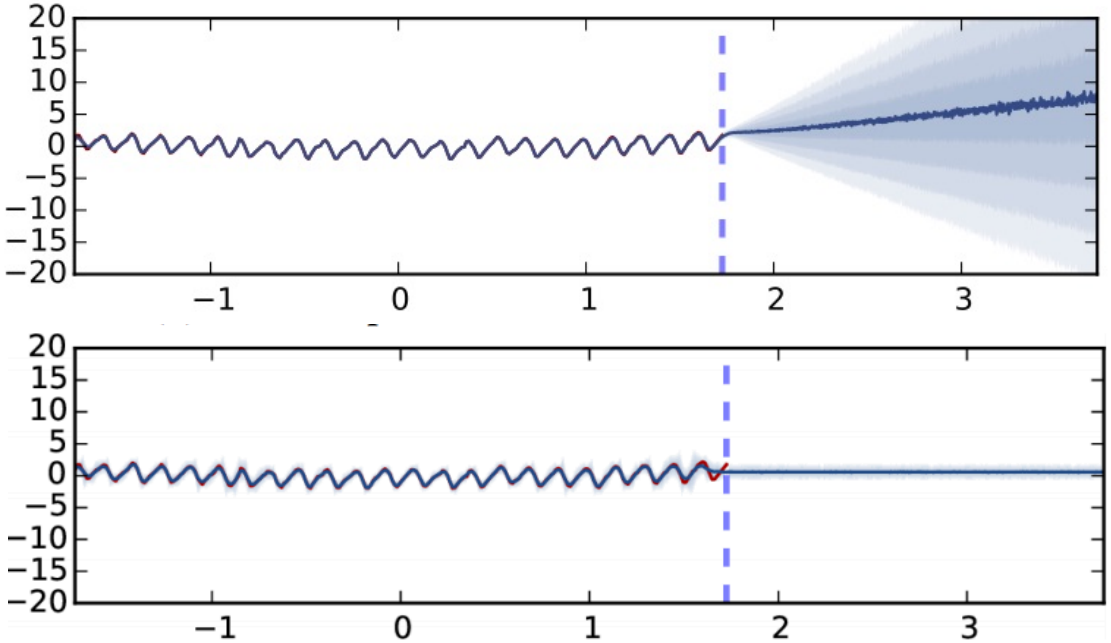
Approximate the geometric mean through scaling the weights by p during test time

Source: Goodfellow et al (2016)

Workhorses

- [Dropout as Bayesian Approximation]():





Stochastic forward passes through a pre-existing architecture

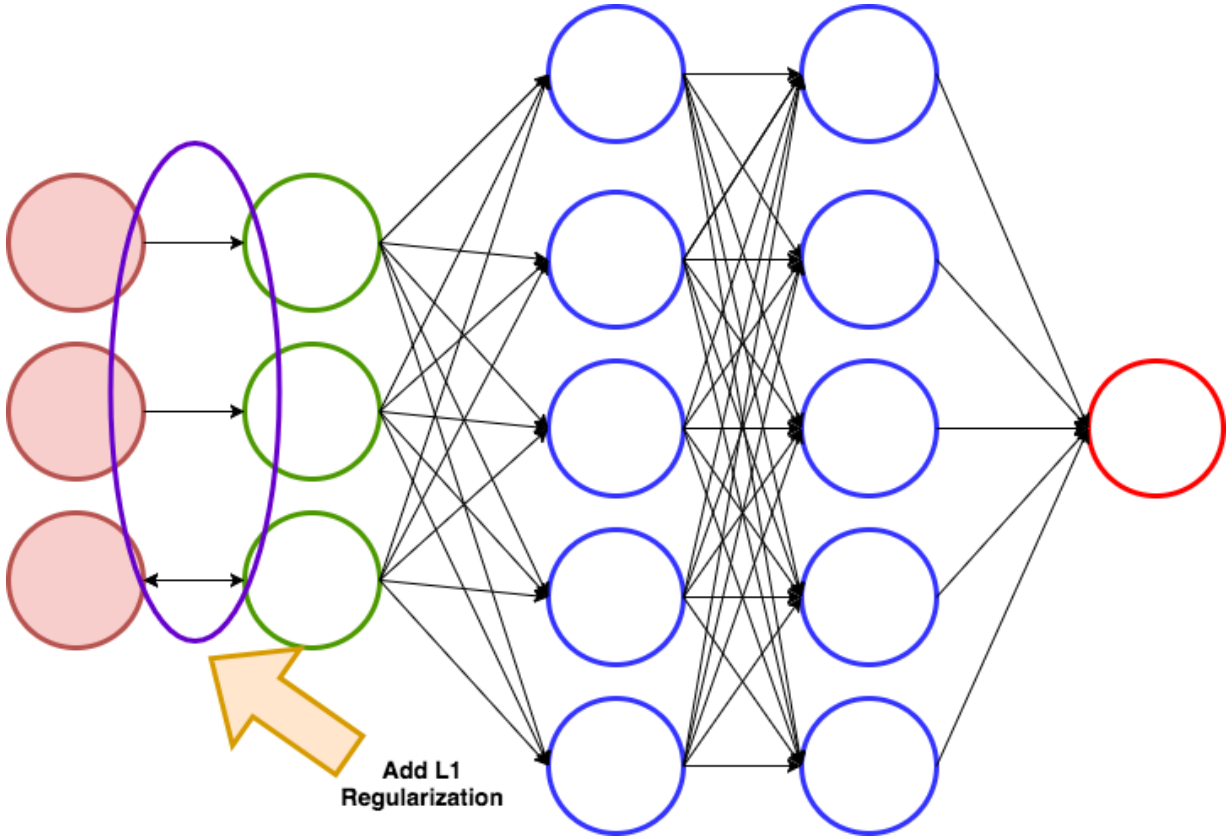
Theoretically produces reasonably calibrated uncertainty estimates for some problems

Practically – doesn't produce nearly enough noise

A simple experiment proves this quite effectively

Clever Hacks

- L1 Layer (Madeka et al [2017]):



Usually through the kitchen sink at a deep model and let it figure it all out for you (with regularization)

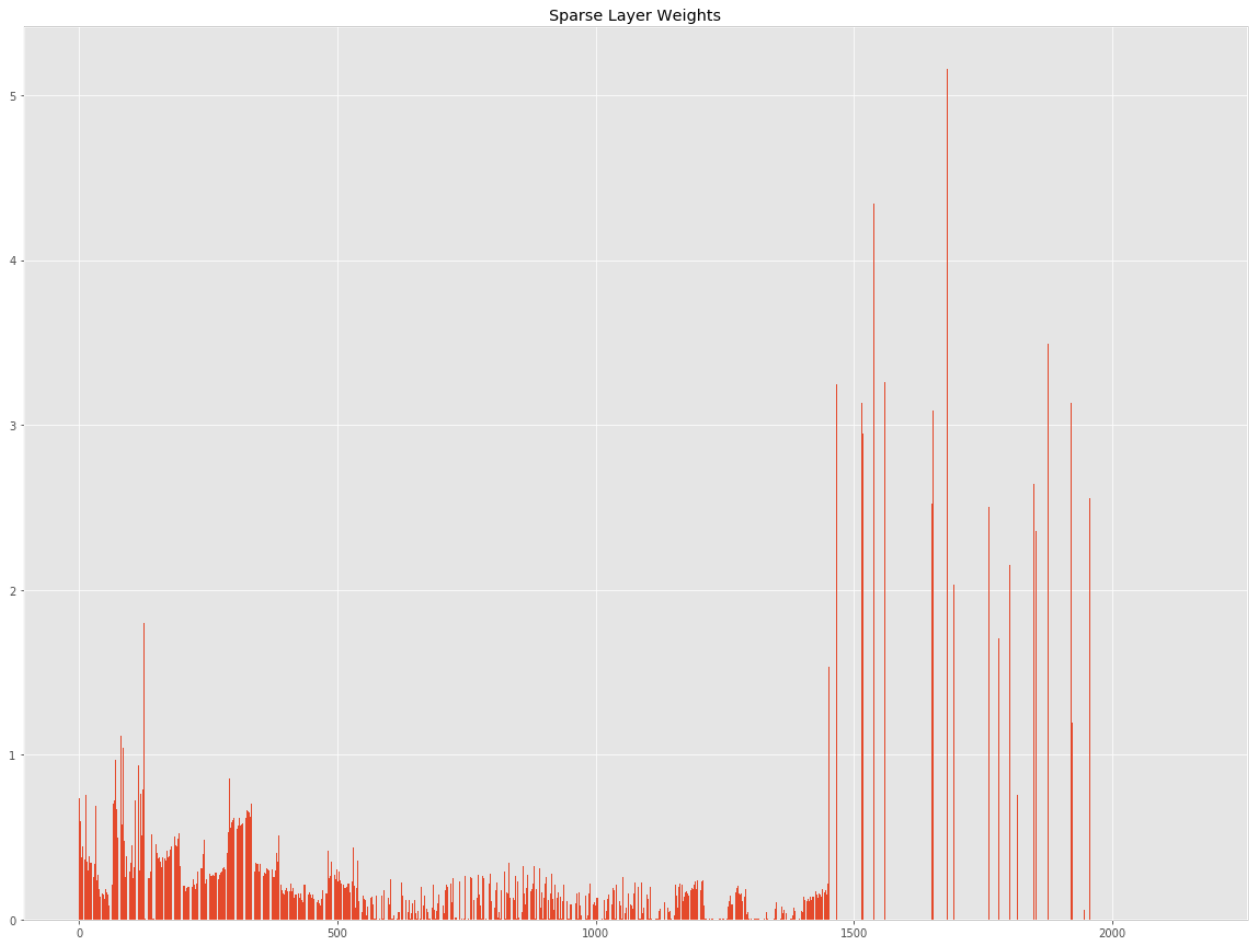
Doesn't really work, and typically makes the whole thing really slow

Another option is to penalize the L1 Norm of the weights for a single layer at the input stage with 1-1 connections with the inputs - $\widehat{x_i} = w_i^1 x_i$

Loss Function - $\mathrm{L}(\mathrm{w}^1, \mathrm{w}, \mathrm{x}) + \lambda \left|\left|w^1\right|\right|_1$

Clever Hacks

- L1 Layer (Madeka et al [2017]):



Most features have 0 weight

Can either do 2 phase learning (retraining without the L1 Layer and the removed features) or just feed 0's to your model in the data iterator

Stops the overhead of a large amount of data

Improves performance by about $> 1.5\sigma$

A Quick Digression on Optimization

What do batch norm and the selu activation do?

They normalize the outputs of each layer to have mean 0 and variance 1!

The equations (for Batch Norm) are:

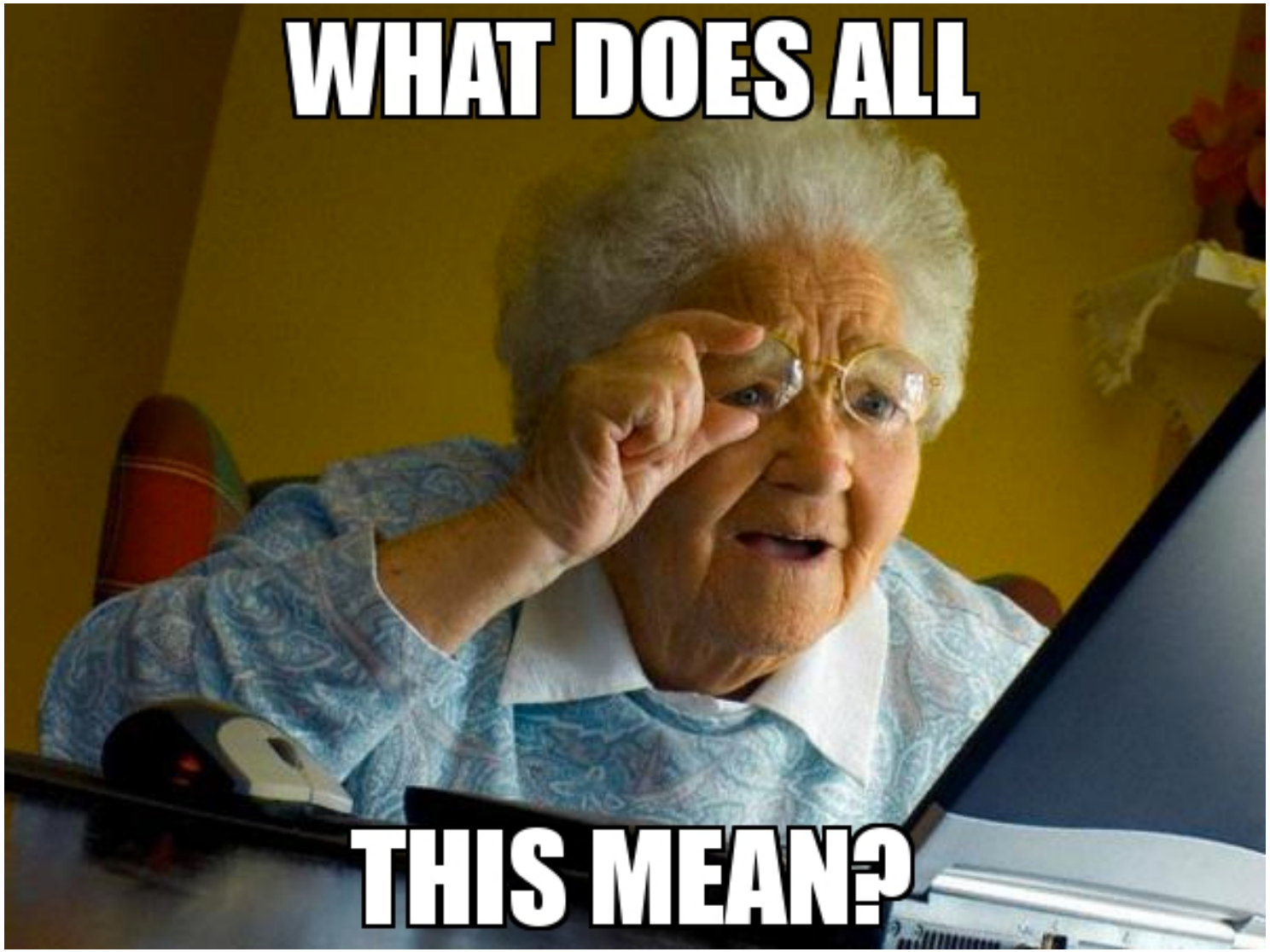
$$\widehat{x^k} = \frac{x^k - \mathbb{E}[x^k]}{\sqrt{Var[x^k]}}$$

Batch norms add a learnable shift and scale:

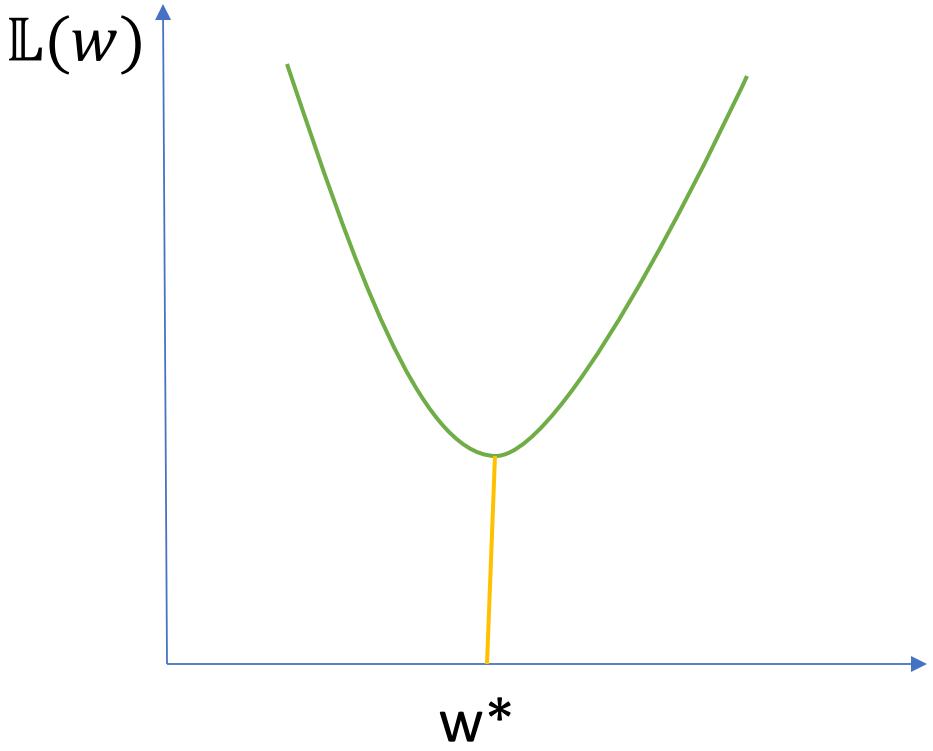
$$y^k = \gamma^k \widehat{x^k} + \beta^k$$

What does it mean?

Dhruv



What does it mean?



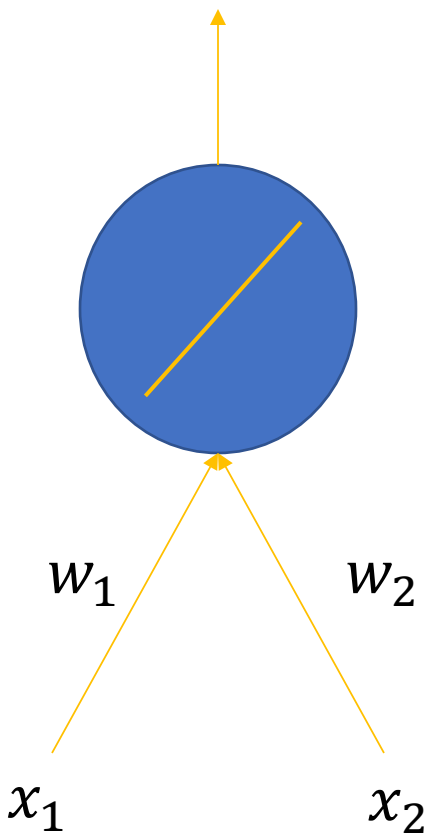
$$L(w) = \frac{1}{2}a(w - w^*)^2 + \mathbb{L}(w^*)$$

$$\Longrightarrow \frac{\partial \mathbb{L}(w_0)}{\partial w} = a(w_0 - w^*)$$

$$\Longrightarrow \frac{\partial^2 \mathbb{L}(w)}{\partial w^2} = a = \frac{\frac{\partial \mathbb{L}(w_0)}{\partial w}}{w_0 - w^*}$$

$$\Longrightarrow w^* = w_0 - \left(\frac{\partial^2 \mathbb{L}(w)}{\partial w^2}\right)^{-1} \frac{\partial \mathbb{L}(w_0)}{\partial w}$$

What does it mean?



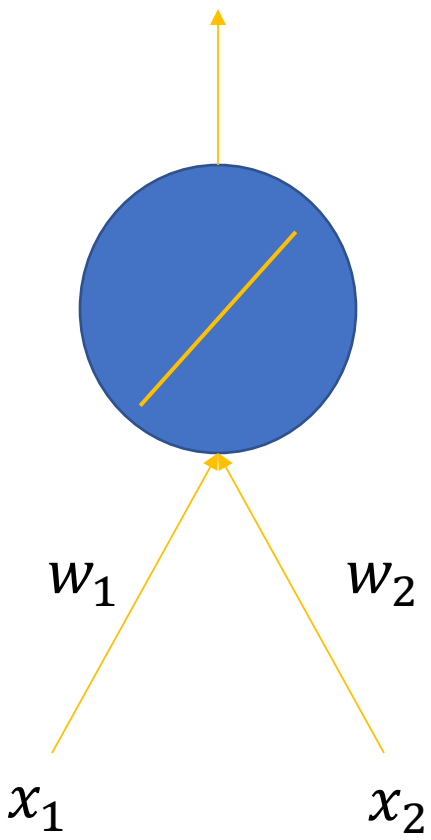
$$\mathbb{L}(w) = \frac{1}{p}\sum_{i=1}^{P}\frac{1}{2}(y_i - w^T x_I)^2$$

$$\frac{\partial \mathbb{L}(w)}{\partial w} = -\frac{1}{p}\sum_{i=1}^{P}(y_i - w^T x_i)x_i$$

$$\frac{\partial^2 \mathbb{L}(w)}{\partial w \partial w^T} = \frac{1}{p}XX^T$$

The second derivative is the covariance matrix of the input samples (decentered)

What does it mean?



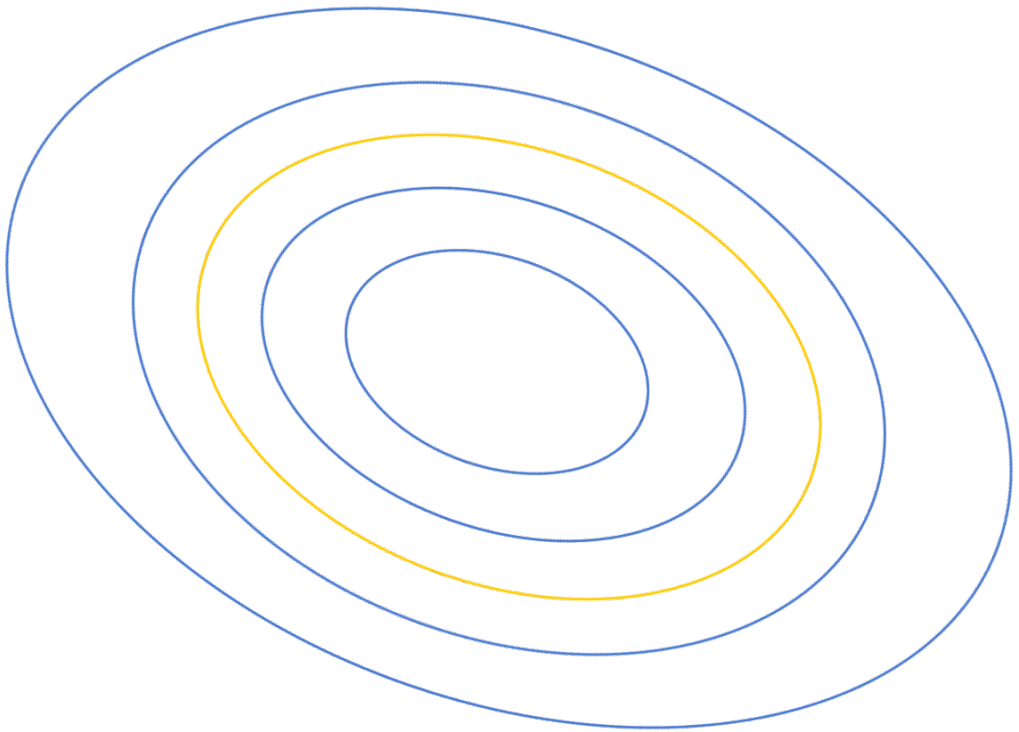
$$\mathbb{L}(w) = \mathbb{L}(w^*) + \frac{1}{2}(w - w^*)^T H (w - w^*)$$

$$\rightarrow \mathbb{L}(w) = \mathbb{L}(w^*) + \frac{1}{2}(w - w^*)^T M^T \Lambda M (w - w^*)$$

$$\rightarrow L(u) = L(u^*) + \frac{1}{2}(u - u^*)^T \Lambda (u - u^*) \; [where\ U = MW]$$

U is the coordinates of w in the eigenspace of H

What does it mean?



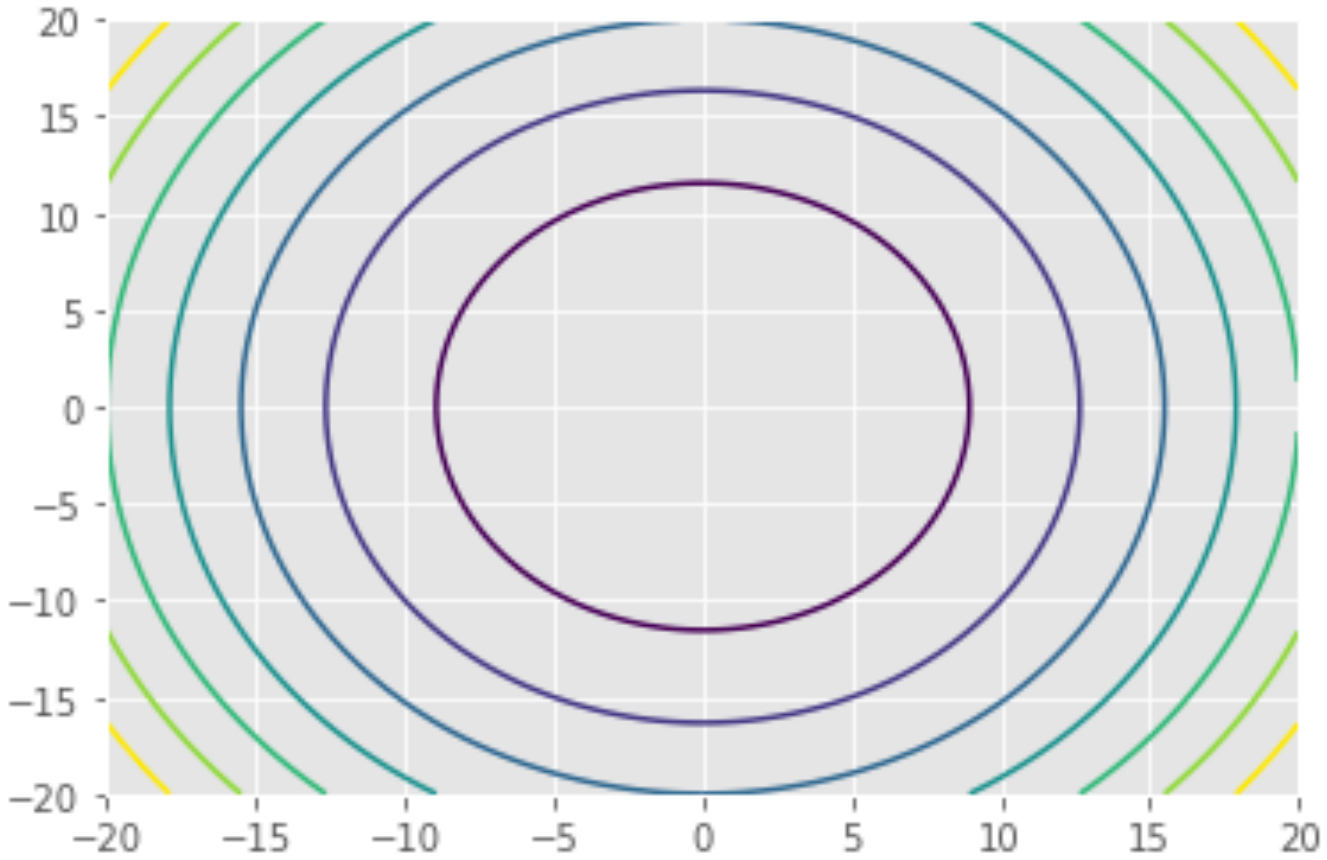
Gradient

$$U_{t+1} = U_t + \frac{1}{2}\Lambda^{-1}\Lambda(U_t - U^*)$$

$$\rightarrow U_{t+1} = U^*$$

$$U_{t+1} = U_t - \Lambda^{-1}\left(\frac{\partial \mathbb{L}}{\partial U}\right)$$

What does it mean?



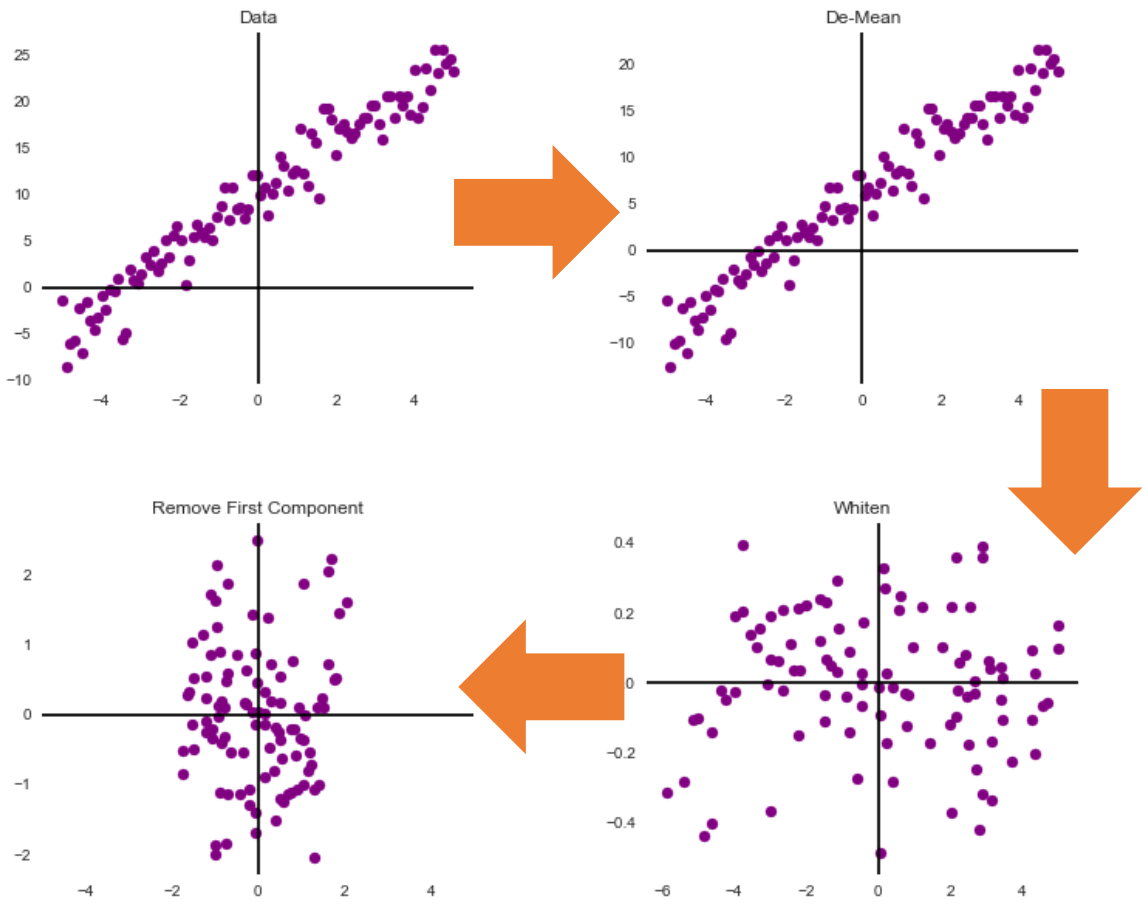
When H is diagonal, principal axes are aligned with the frame of reference.

$$H = \lambda I$$

Some tricks

A few things to keep in mind when you actually train networks:

- Shuffle the batches so successive batches are very rarely homogeneous
- Present input examples that have large error more often than ones that have small error
- Normalize inputs so that the hessian behaves well, if you can - try to whiten them

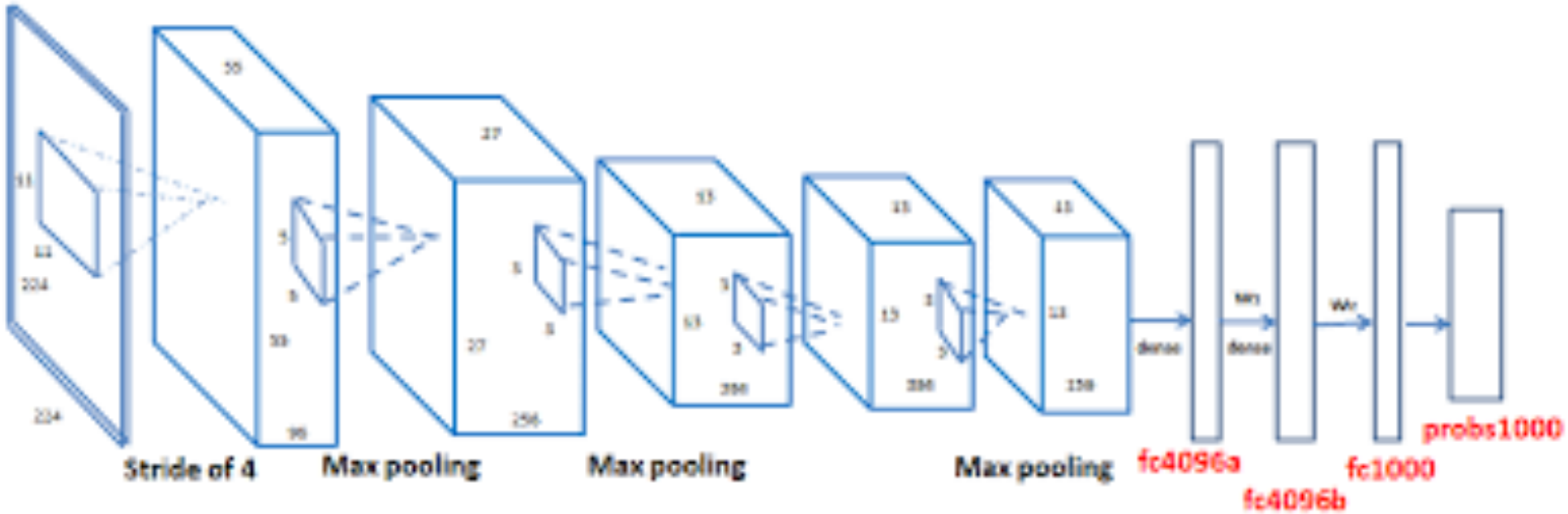


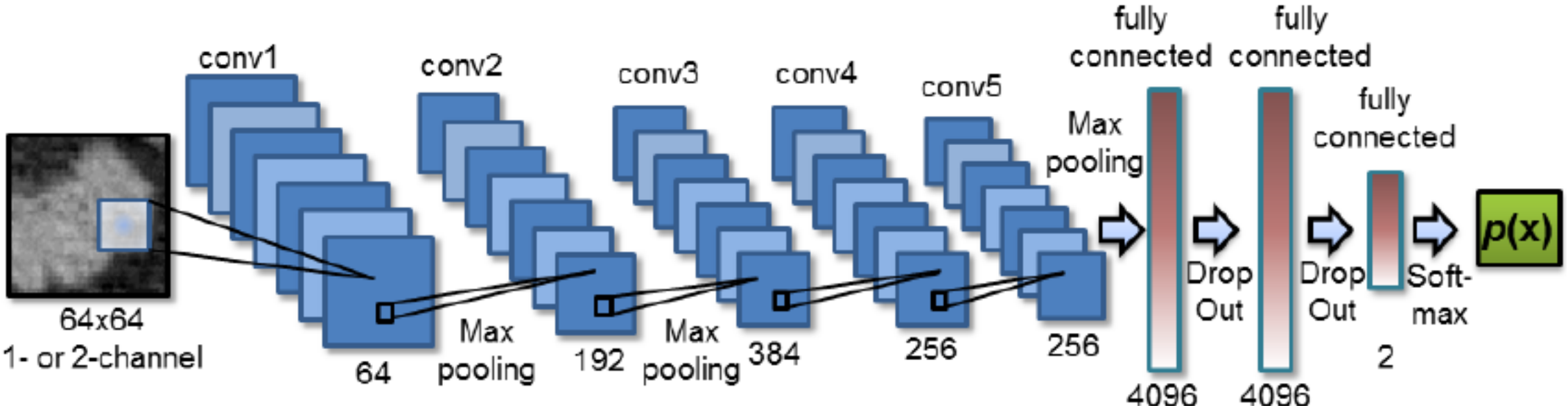
Convolutional Neural Networks

What are ConvNets?

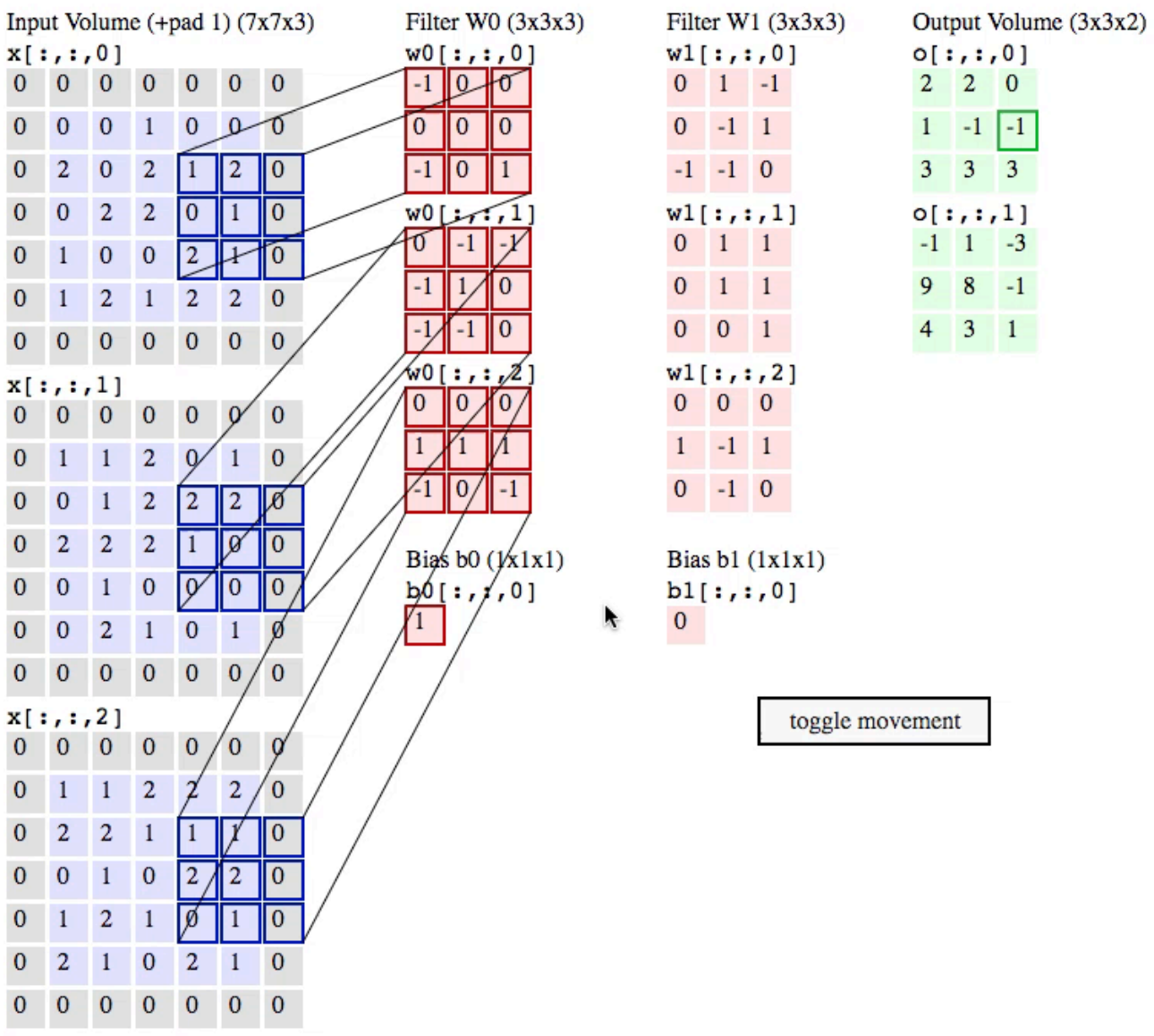
- Patterns in images tend to be compositional – very much locally correlated but not so much for far away pixels
- We can build this into our network architecture for images
- The idea of replicating a detector on the entire visual field is the same idea as replicating a detector in time

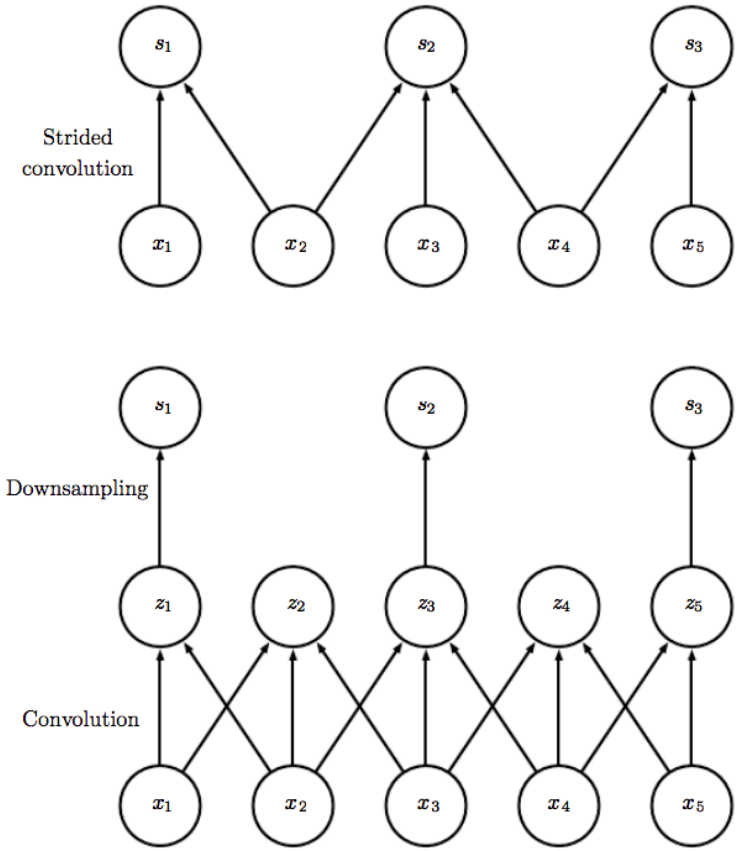
Visual Architectures





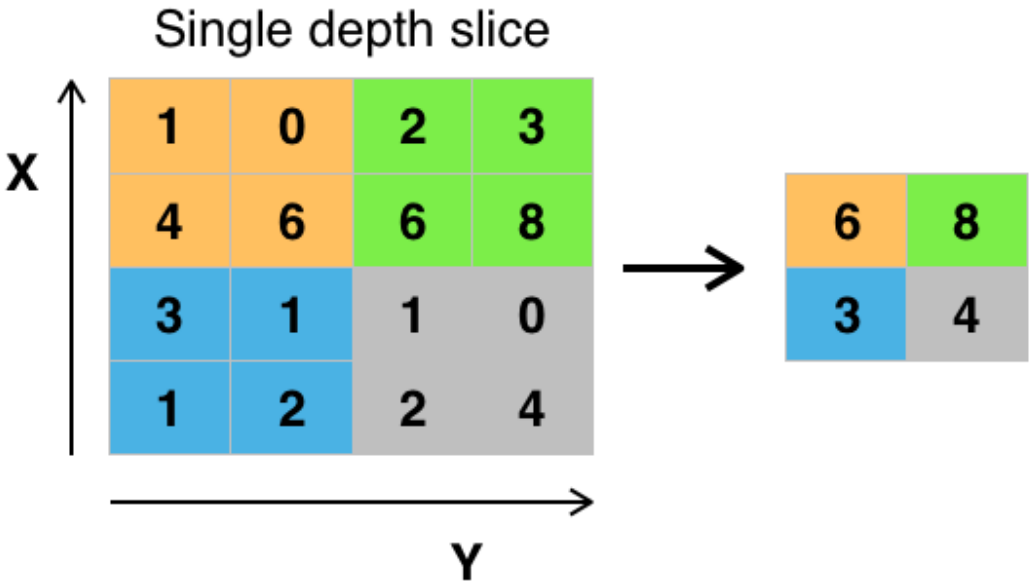
Convolutional Kernels

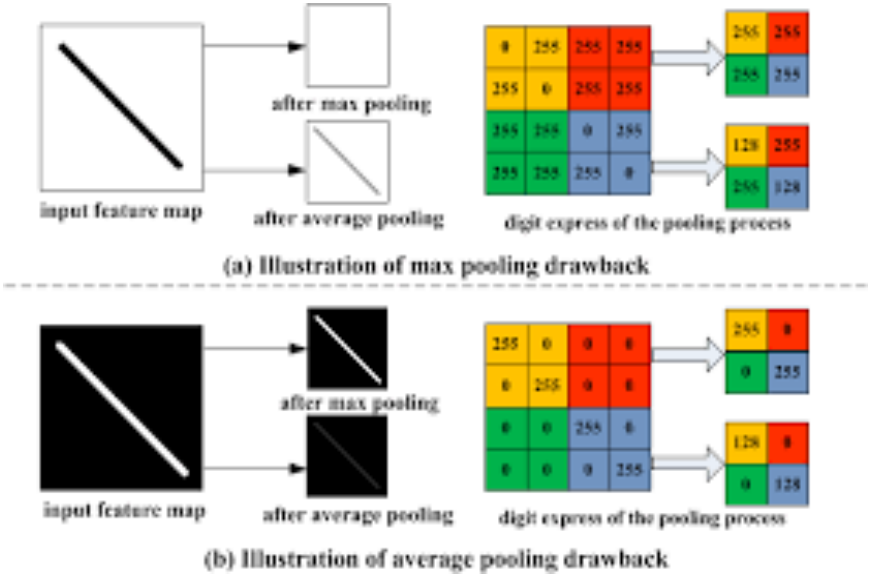


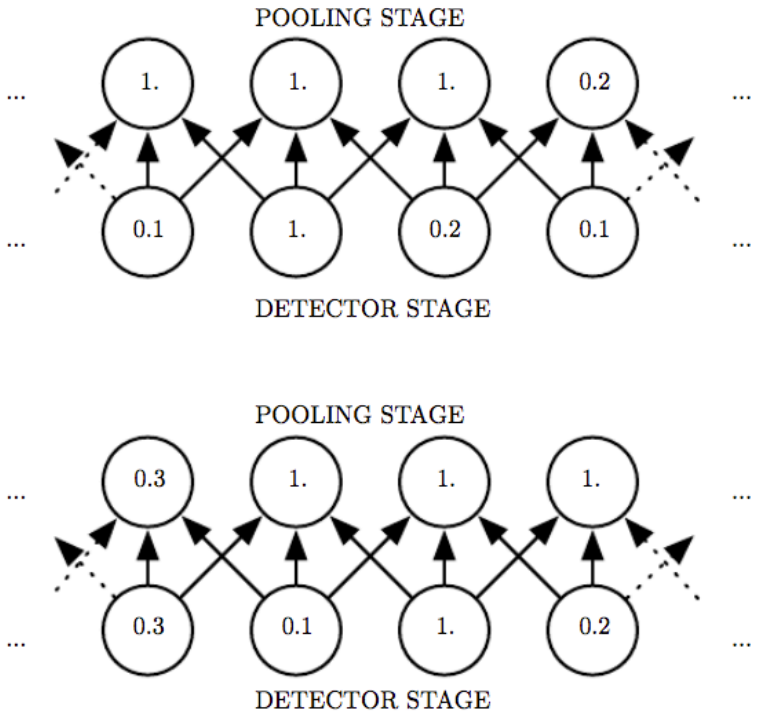


- Parameter sharing greatly reduces number of free parameters
- Don't really see CV tasks without ConvNets

Pooling Layers



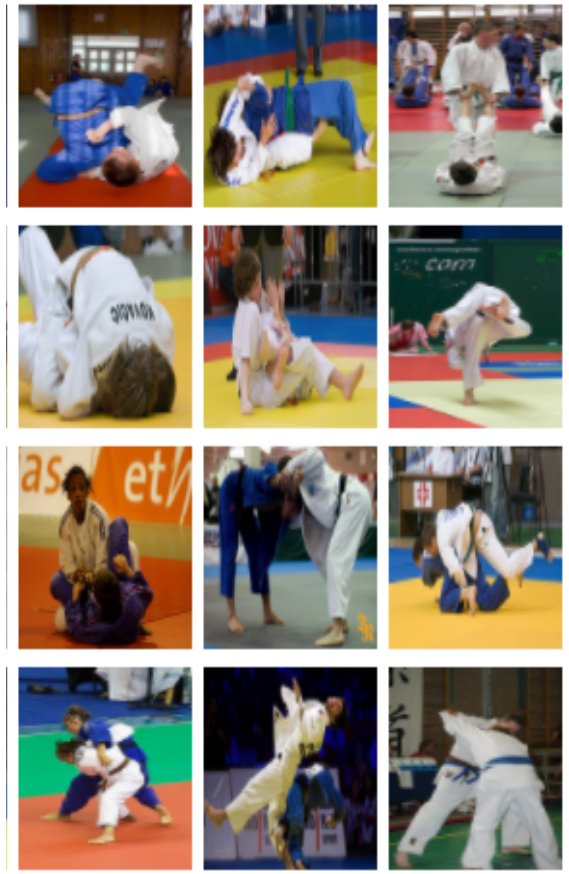


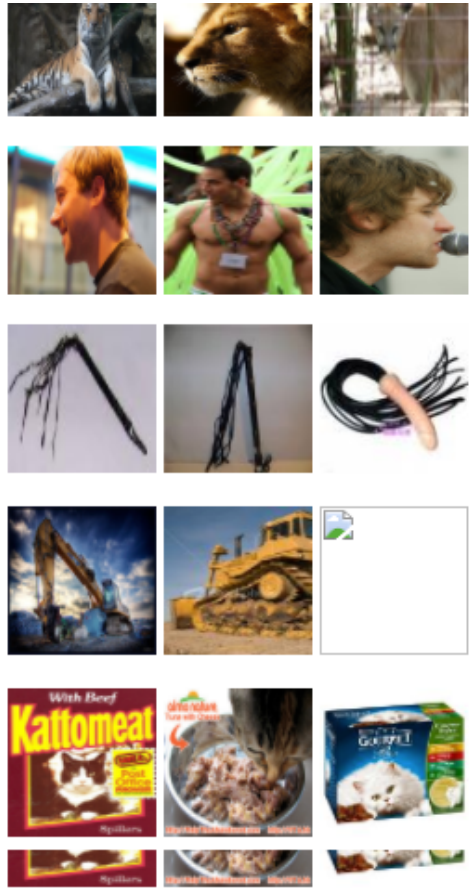


- Subsampling layer that builds local invariances into the structure
- Backpropagation typically passes a gradient of 1 to the cell where the max occurred and 0 everywhere else
- You see less and less of it – we do not want the neural activities to be invariant to the view point we want them to be changed by the change in viewpoint
- Fail to use an underlying linear manifold where the variance of the image lies

A little history - ImageNet

- ImageNet [Fei-Fei Li et al 2014] – 1.5 million images, 1000 Categories





A little history - ImageNet

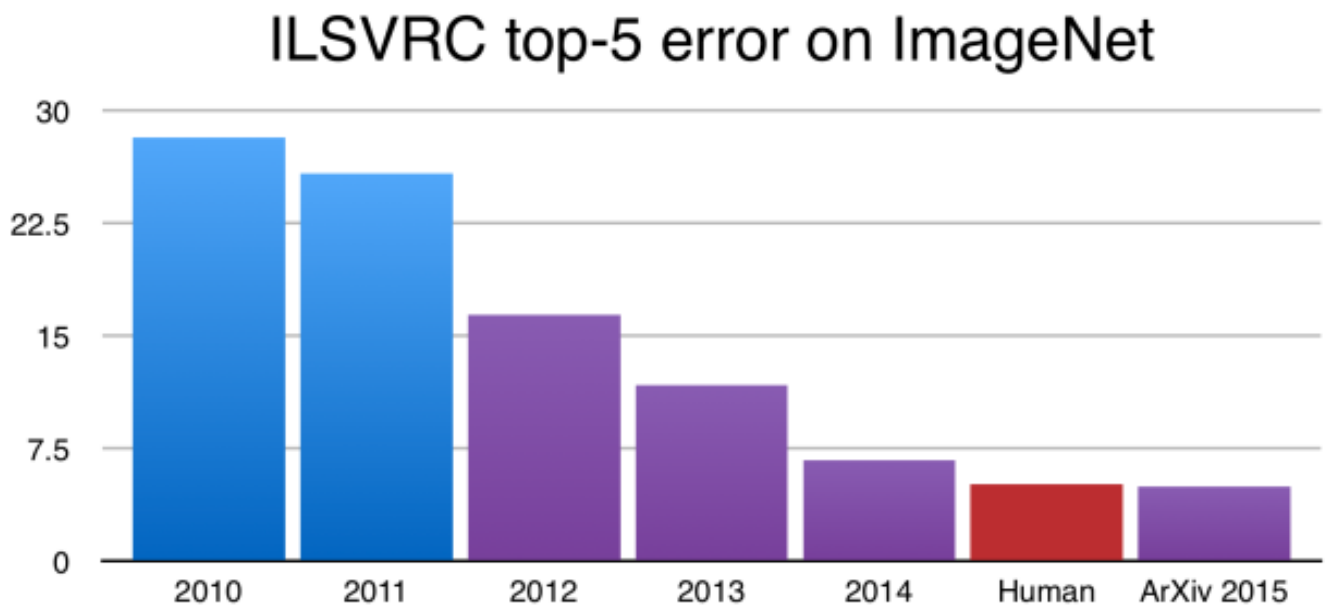
- ImageNet [Fei-Fei Li et al 2014] – 1.5 million images, 1000 Categories

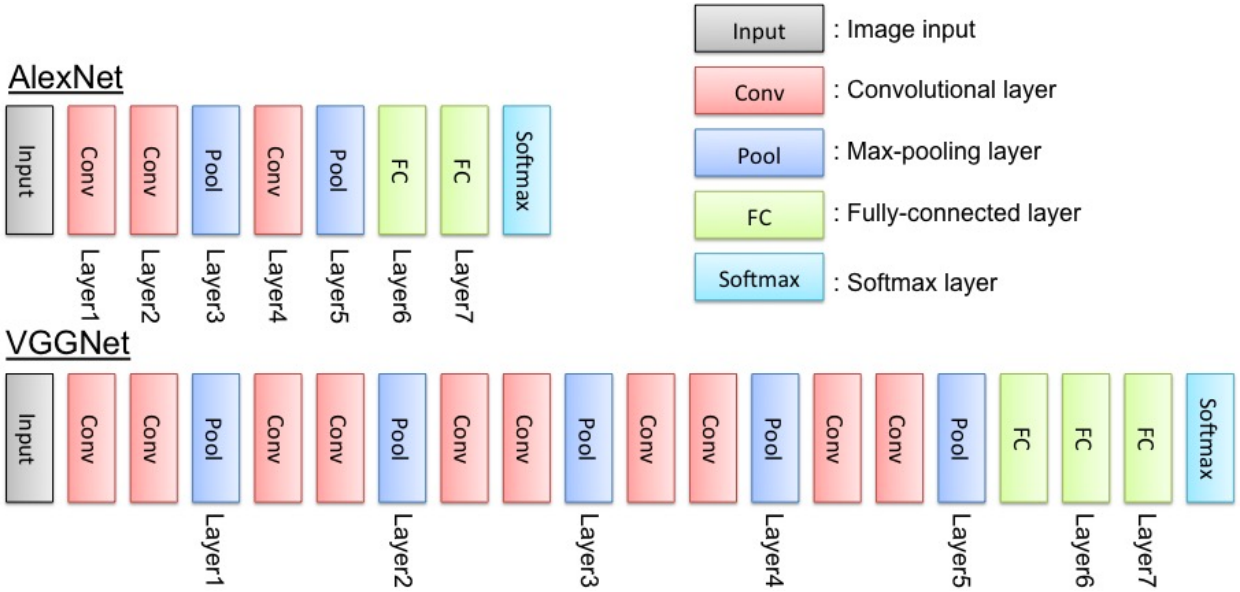
Female Mammal

Anthropod

Young bird

ImageNet





Post AlexNet – almost all major ImageNet submissions have been ConvNets.

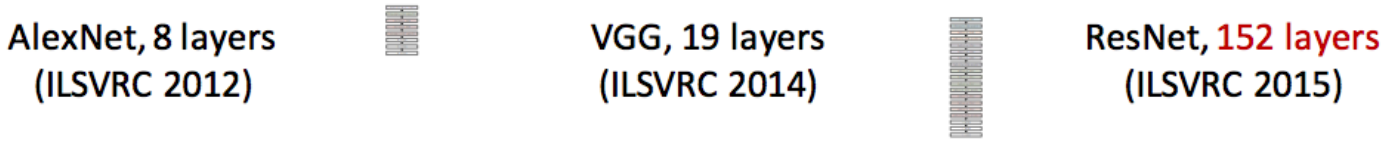
AlexNet was the first fast large scale GPU implementation of ConvNets

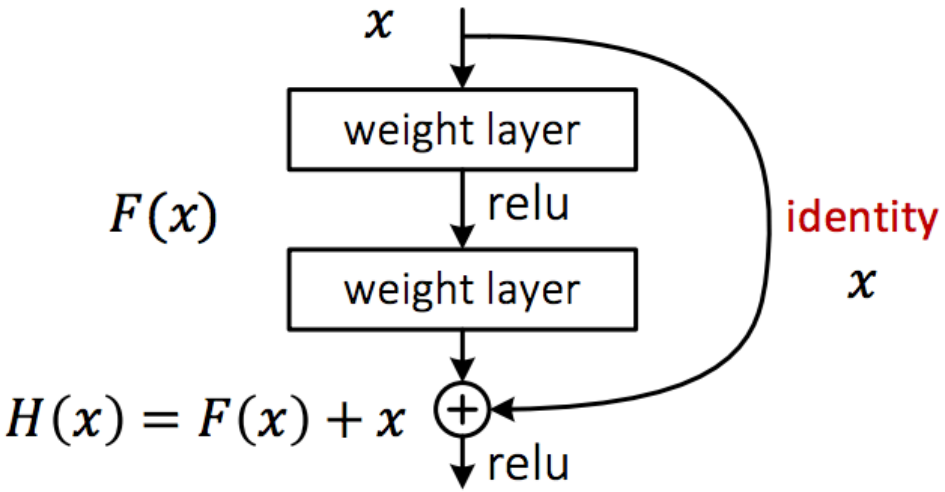
Deeper networks, longer training cut the errors by another factor (2014) of 2

VGGNet (from Oxford) showed that actually you want smaller kernels but many non-linearities

```
num_fc = 512
net = gluon.nn.Sequential()
with net.name_scope():
    net.add(gluon.nn.Conv2D(channels=20, kernel_size=5, activation='relu'))
    net.add(gluon.nn.MaxPool2D(pool_size=2, strides=2))
    net.add(gluon.nn.Conv2D(channels=50, kernel_size=5, activation='relu'))
    net.add(gluon.nn.MaxPool2D(pool_size=2, strides=2))
    # The Flatten layer collapses all axis, except the first one, into one axis.
    net.add(gluon.nn.Flatten())
    net.add(gluon.nn.Dense(num_fc, activation="relu"))
    net.add(gluon.nn.Dense(num_outputs))
```

ResNet





$$x_L = x_l + \sum_{i=l}^{L-1} F(x_i)$$

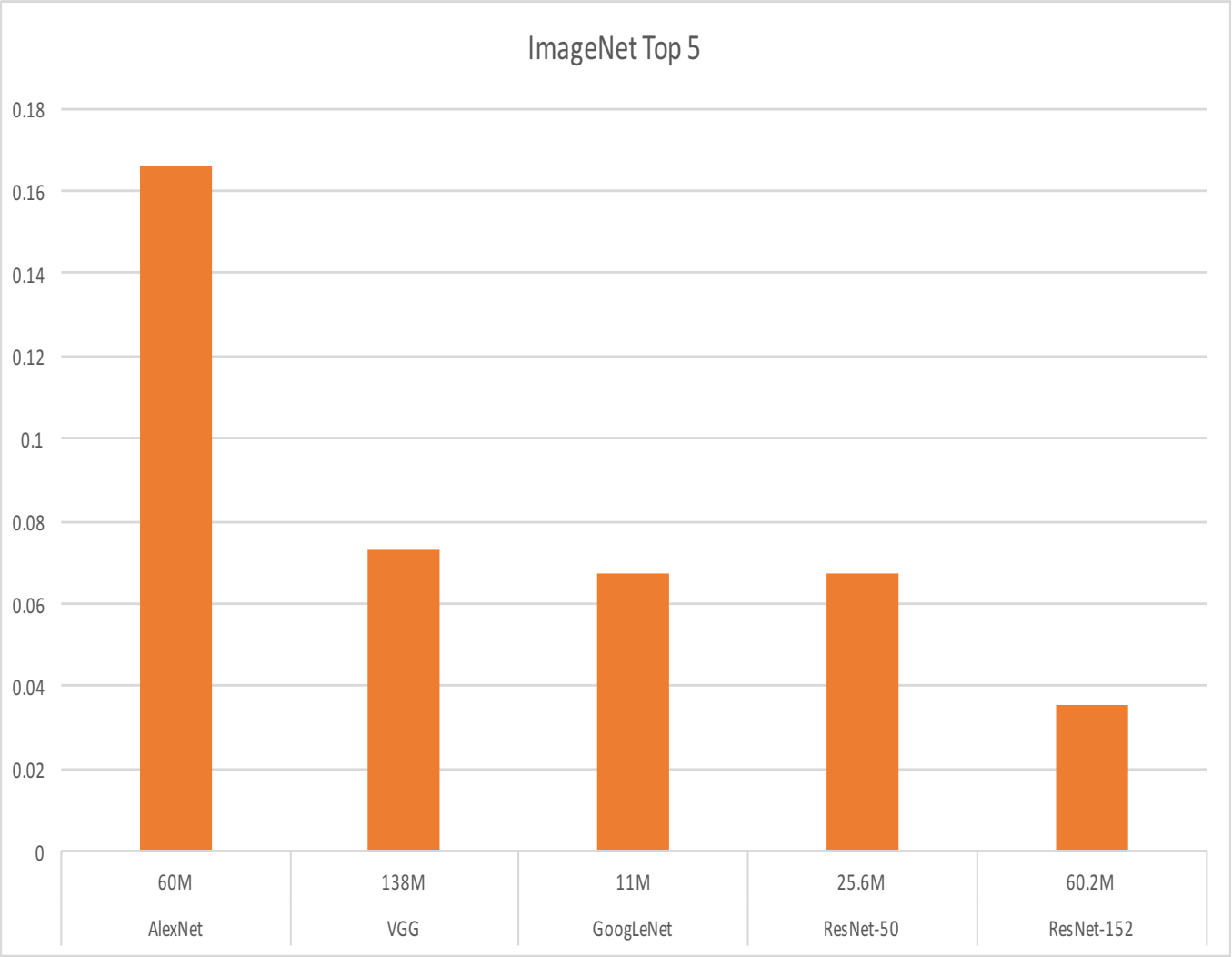
$$\frac{\partial E}{\partial x_l} = \frac{\partial E}{\partial x_L}\frac{\partial x_L}{\partial x_l} = \frac{\partial E}{\partial x_L}\left(1 + \frac{\partial}{\partial x_l}\sum_{i=1}^{L-1} F(x_i)\right)$$

$$\frac{\partial E}{\partial x_l}$$

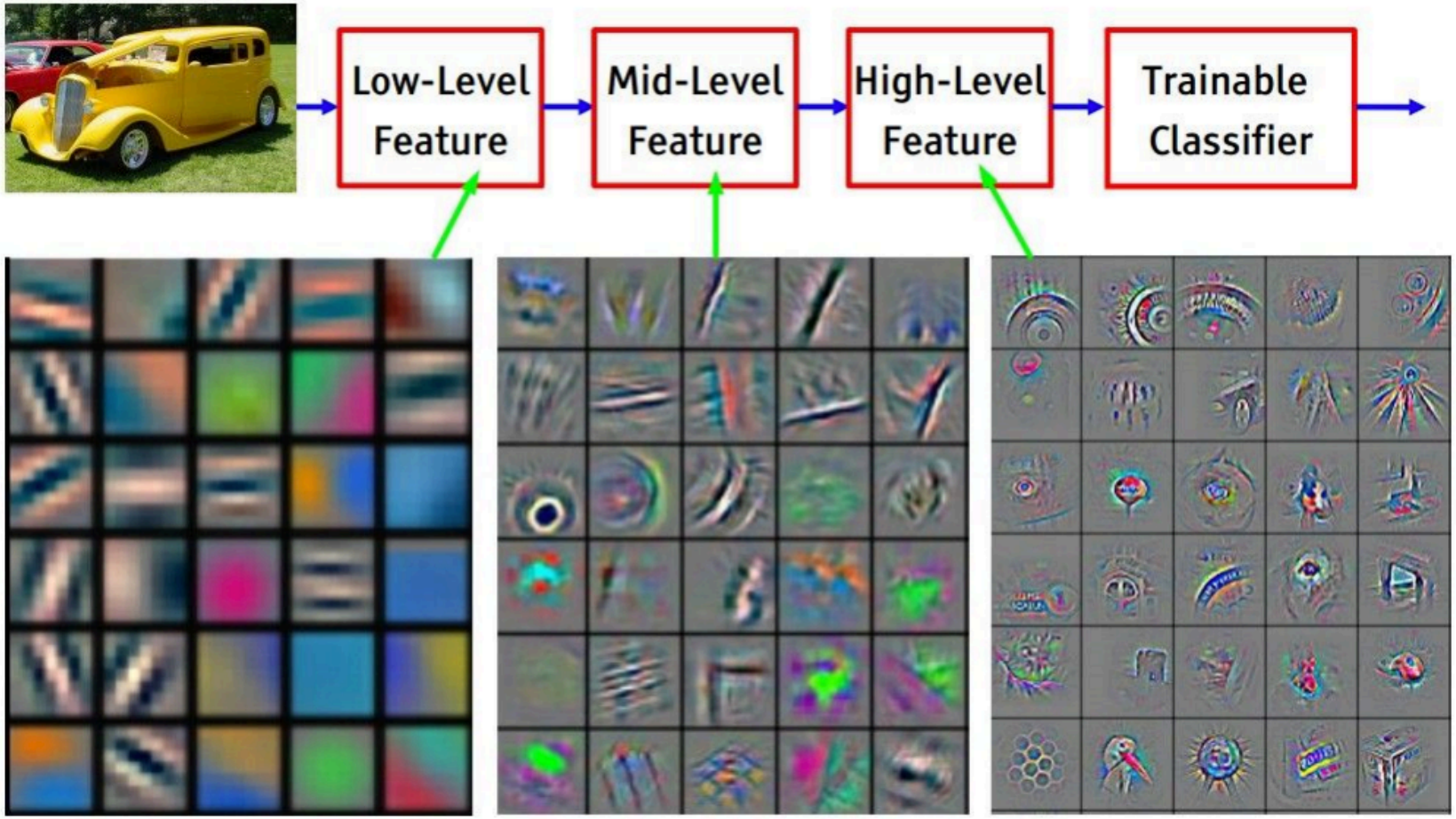
$$\frac{\partial E}{\partial x_L}$$

Advantages of Depth

Model	Number of free parameters	ImageNet Top 5
AlexNet	60M	83.4%
VGG	138M	92.7%
GoogLeNet	11M	93.3%
ResNet-50	25.6M	93.3%
ResNet-152	60.2M	96.43%



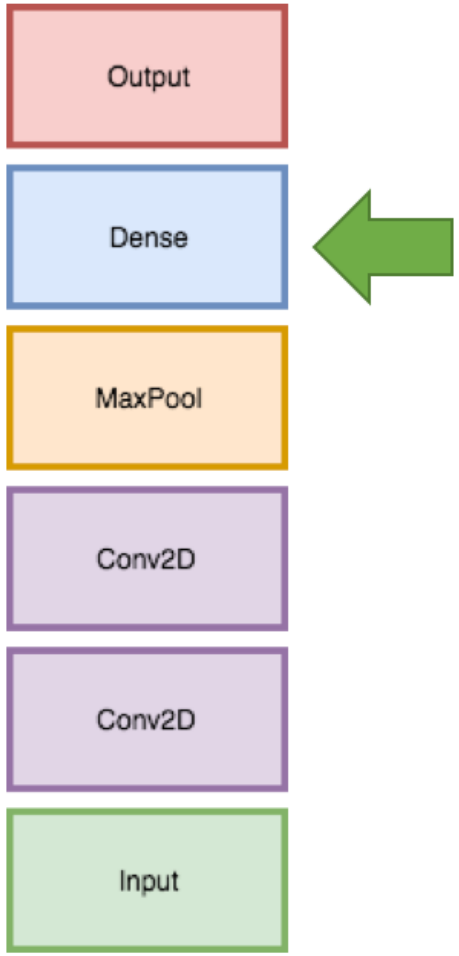
What do ConvNets really learn?



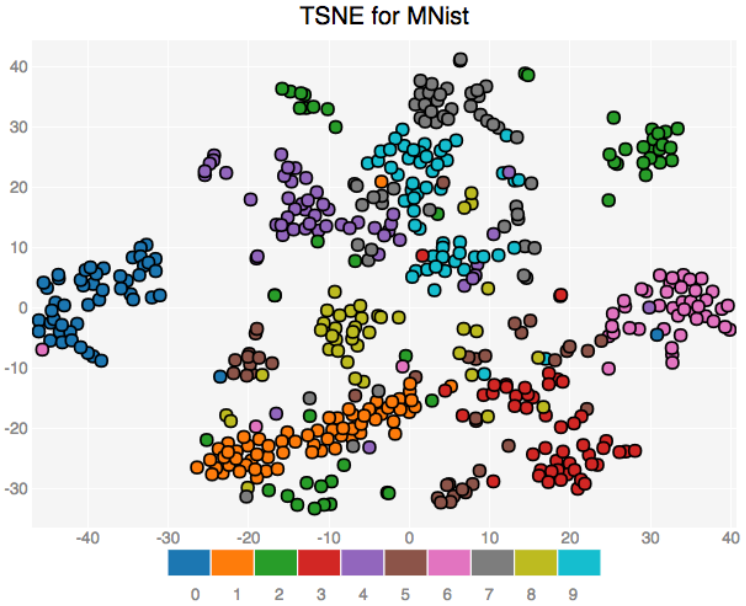
Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

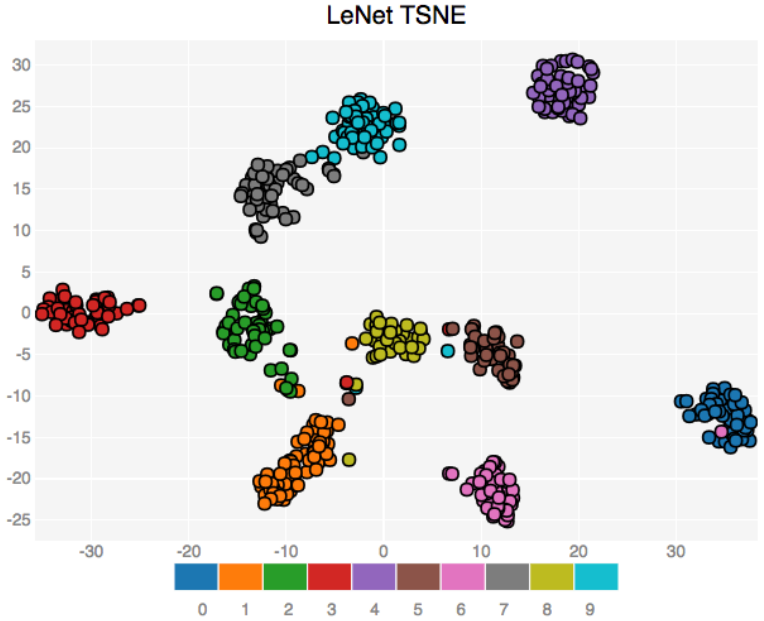
Source: LeCun (2016)

What do ConvNets really learn?



Let's look at the representations learned by this layer





Recurrent Neural Networks

What do we need to learn a function?

- Imagine if we want a function that counts the number of non-zero's in a vector $(x_1, \dots, x_N)$
- Ingredients:
  - Memory s <- counts number of positive elements
  - Function `func` applied to each element one at a time updates the memory
- A computer takes each element of a sequence of instructions and manipulates the registry
- Registry is the memory, cpu instruction is the function

```
s = 0

def func(v, s):
    if v <= 0:
        return s
    else:
        return s+1

for i in x:
    s = func(i, s)
```

Recurrent Neural Networks

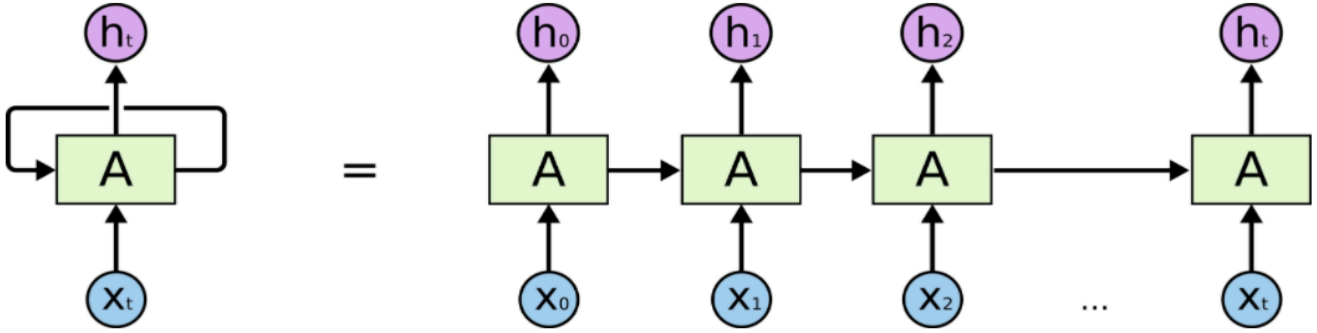
- Parametric recursive function
- Has a memory h
- Applies a function g to each element of an input x_i

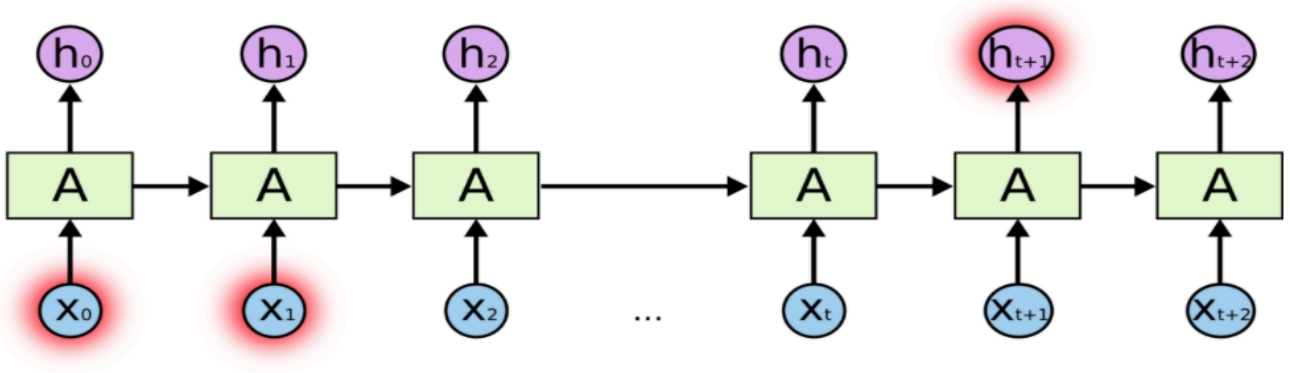
$$h_t = f(x_t, h_{t-1})$$

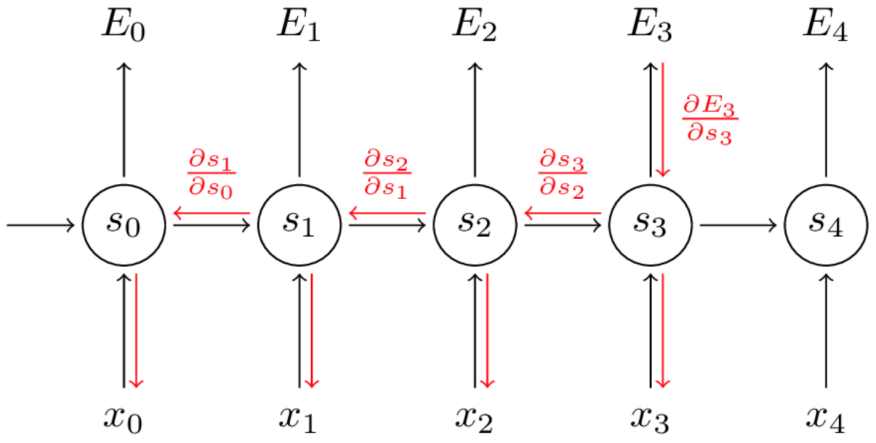
$$f(x_t, h_{t-1}) = g(Wx_t + Uh_{t-1})$$

- BPTT: $c(x) := g\big(f_1(x), \dots, f_N(x)\big)$

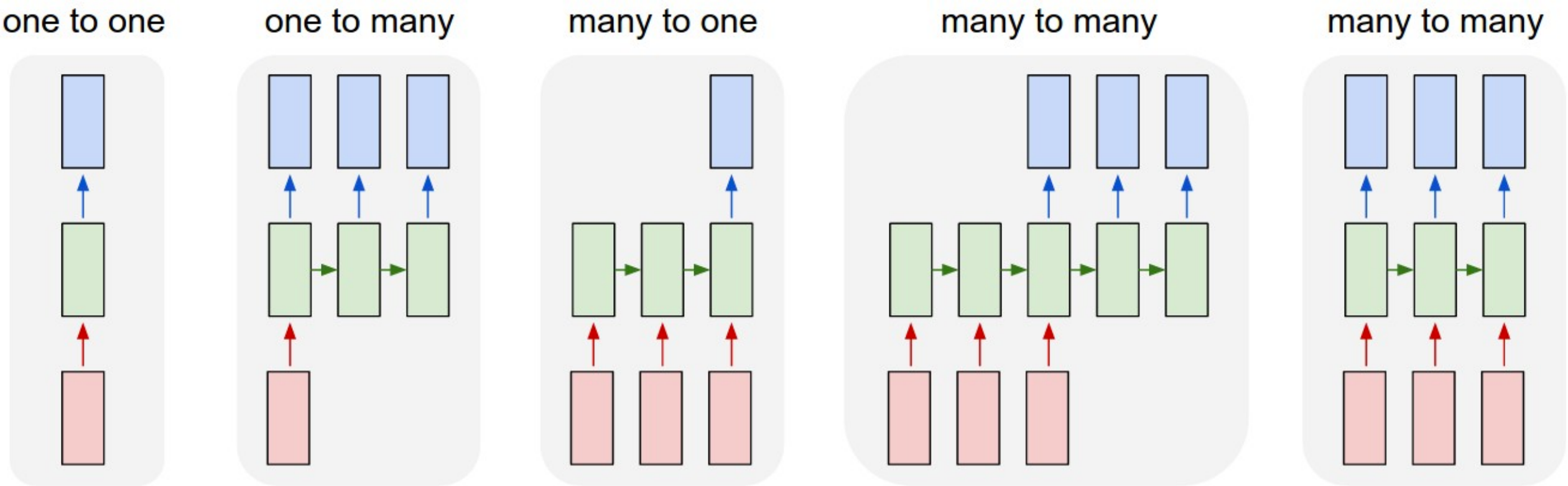
$$\frac{\partial c}{\partial x} = \frac{\partial c}{\partial g} \sum_{i=1}^{N} \frac{\partial f_i}{\partial x}$$







RNN’s for Sequence Modelling



Source: [Karpathy (2015)]

The Unreasonable Effectiveness of RNNs

For $\bigoplus_{n=1,\ldots,m}$ where $\mathcal{L}_{m_\bullet} = 0$, hence we can find a closed subset $\mathcal{H}$ in $\mathcal{H}$ and any sets $\mathcal{F}$ on X, U is a closed immersion of S, then $U \to T$ is a separated algebraic space.

Proof. Proof of (1). It also start we get

$$S = \mathrm{Spec}(R) = U \times_X U \times_X U$$

and the compariccly in the fibre product covering we have to prove the lemma generated by $\coprod Z \times_U U \to V$. Consider the maps M along the set of points Sch_{fppf} and $U \to U$ is the fibre category of S in U in Section, ?? and the fact that any U affine, see Morphisms, Lemma ??. Hence we obtain a scheme S and any open subset $W \subset U$ in $Sh(G)$ such that $\mathrm{Spec}(R') \to S$ is smooth or an

$$U = \bigcup U_i \times_{S_i} U_i$$

which has a nonzero morphism we may assume that f_i is of finite presentation over S. We claim that $\mathcal{O}_{X,x}$ is a scheme where $x, x', s'' \in S'$ such that $\mathcal{O}_{X,x'} \to \mathcal{O}'_{X',x'}$ is separated. By Algebra, Lemma ?? we can define a map of complexes $\mathrm{GL}_{S'}(x'/S'')$ and we win. $\square$

To prove study we see that $\mathcal{F}|_U$ is a covering of $\mathcal{X}'$, and $\mathcal{T}_i$ is an object of $\mathcal{F}_{X/S}$ for $i > 0$ and $\mathcal{F}_p$ exists and let $\mathcal{F}_i$ be a presheaf of $\mathcal{O}_X$-modules on $\mathcal{C}$ as a $\mathcal{F}$-module. In particular $\mathcal{F} = U/\mathcal{F}$ we have to show that

$$\widetilde{M}^\bullet = \mathcal{I}^\bullet \otimes_{\mathrm{Spec}(k)} \mathcal{O}_{S,s} - i_X^{-1}\mathcal{F})$$

is a unique morphism of algebraic stacks. Note that

$$\mathrm{Arrows} = (Sch/S)^{opp}_{fppf}, (Sch/S)_{fppf}$$

and

$$V = \Gamma(S, \mathcal{O}) \longmapsto (U, \mathrm{Spec}(A))$$

is an open subset of X. Thus U is affine. This is a continuous map of X is the inverse, the groupoid scheme S.

Proof. See discussion of sheaves of sets. $\square$

The result for prove any open covering follows from the less of Example ??. It may replace S by $X_{spaces,étale}$ which gives an open subspace of X and T equal to S_{Zar}, see Descent, Lemma ??. Namely, by Lemma ?? we see that R is geometrically regular over S.

Lemma 0.1. *Assume (3) and (3) by the construction in the description.*

Suppose $X = \lim |X|$ (by the formal open covering X and a single map $\underline{Proj}_X(\mathcal{A}) = \mathrm{Spec}(B)$ over U compatible with the complex

$$Set(\mathcal{A}) = \Gamma(X, \mathcal{O}_{X,\mathcal{O}_X}).$$

When in this case of to show that $\mathcal{Q} \to \mathcal{C}_{Z/X}$ is stable under the following result in the second conditions of (1), and (3). This finishes the proof. By Definition ?? (without element is when the closed subschemes are catenary. If T is surjective we may assume that T is connected with residue fields of S. Moreover there exists a closed subspace $Z \subset X$ of X where U in X' is proper (some defining as a closed subset of the uniqueness it suffices to check the fact that the following theorem

(1) *f is locally of finite type. Since $S = \mathrm{Spec}(R)$ and $Y = \mathrm{Spec}(R)$.*

Proof. This is form all sheaves of sheaves on X. But given a scheme U and a surjective étale morphism $U \to X$. Let $U \cap U = \coprod_{i=1,\ldots,n} U_i$ be the scheme X over S at the schemes $X_i \to X$ and $U = \lim_i X_i$. $\square$

The following lemma surjective restrocomposes of this implies that $\mathcal{F}_{x_0} = \mathcal{F}_{x_0} = \mathcal{F}_{\mathcal{X},\ldots,0}$.

Lemma 0.2. *Let X be a locally Noetherian scheme over S, $E = \mathcal{F}_{X/S}$. Set $\mathcal{I} = \mathcal{J}_1 \subset \mathcal{I}'_n$. Since $\mathcal{I}^n \subset \mathcal{I}^n$ are nonzero over $i_0 \leq \mathfrak{p}$ is a subset of $\mathcal{J}_{n,0} \circ \overline{A}_2$ works.*

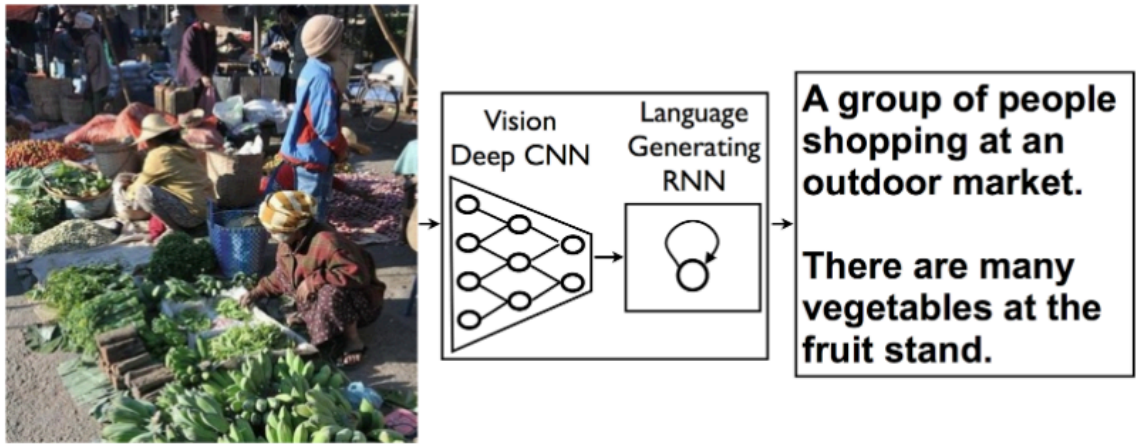
Lemma 0.3. *In Situation ??. Hence we may assume $\mathfrak{q}' = 0$.*

Proof. We will use the property we see that $\mathfrak{p}$ is the mext functor (??). On the other hand, by Lemma ?? we see that

$$D(\mathcal{O}_{X'}) = \mathcal{O}_X(D)$$

where K is an F-algebra where δ_{n+1} is a scheme over S

```
def forward(self, inputs, hidden):
    emb = self.drop(self.encoder(inputs))
    output, hidden = self.rnn(emb, hidden)
    output = self.drop(output)
    decoded = self.decoder(output.reshape((-1, self.num_hidden)))
    return decoded, hidden
```



(Vinyals, *et al.* (2014))

Does this actually work?

- You never see Vanilla RNN's – just like you rarely ever see MLP's in practice
- Hyper difficult to train – you update the entire "memory"/"registry" h for every single time step
- A real CPU only updates part of the registry at each time step:

$$h_t = (1 - u) \odot h_{t-1} + u \odot \widetilde{h_t}$$

- A real CPU rarely ever reads all the values of the registry:

$$\widetilde{h_t} = f(x_t, r \odot h_{t-1}) = g(Wx_t + U(r \odot h_{t-1}))$$

Does this actually work?

- Here's where our analogy should end
- u, r cannot be binary for a network – it would create discontinuous functions. We make them real:
  - r – computes how much of the previous hidden state should be used
  - u – computes how much of the memory state is replaced
- In a typical CPU – how much of the registry is used and how much is replaced is hard coded for every operation. In our case, we'd like to learn it

$$r = g(W^r x_t + U^r h_{t-1})$$

$$u = g(W^u x_t + U^u h_{t-1})$$

This Is the famed Gated Recurrent Unit. A simple memory cell that often outperforms its more famous and complex ancestor – the LSTM

Long-Short Term Memory Networks

- Explicitly separates output h and memory c
- Output gates control how much memory is revealed:

$$o = \sigma(W_o x_t + U_o h_{t-1})$$
$$h_t = o \odot \tanh(c_t)$$

- Memory update resembles GRU:

$$c_t = f \odot c_{t-1} + i \odot \widetilde{c_t}$$

- f, t are exactly like the update/reset gates
- The candidate memory state $\widetilde{c_t}$ is computed the same way as $\widetilde{h_t}$ for the GRU

Why is it so hard to train RNN's?

- $g(a) = a$ simplest unbounded element wise non-linearity
- $$\begin{aligned} h_t &= Wx_t + Uh_{t-1} \\ h_t &= Wx_t + U\big(Wx_{t-1} + U(Wx_{t-2} + U \dots)\big) \\ h_t &= Wx_t + UUWx_{t-1} + \cdots \\ h_t &= \sum_{i=1}^{l} \left(\prod_{l'}^{l-t} U \right) Wx_i \end{aligned}$$
- Doesn't look nice does it?

Why is it so hard to train RNN's?

$$\prod_{l'}^{l-t} U = QS^{l-t}Q^{-1}$$

$$\left(\prod_{l'}^{l-t} U\right) Wx_i = \mathrm{diag}\left(\mathrm{S}^{\mathrm{l-t}}\right) \odot Wx_i$$

$$e_{max} = \max diag(S) > 1 \Longrightarrow \left|\left|h_t\right|\right| \Longrightarrow \infty$$

Rectifiers don't work!

What happens if we use a saturated non-linearity?

$$\frac{\partial h_{l1}}{\partial x_{t_0+1}} = \frac{\partial h_{t1}}{\partial h_{(t_0+1)}} \frac{\partial h_{t_0+1}}{\partial x_{(t_0+1)}}$$

$$\frac{\partial h_{t1}}{\partial h_{(t_0+1)}} = \frac{\partial h_{t1}}{\partial h_{(t_1-1)}} \frac{\partial h_{t_1-1}}{\partial h_{(t_1-2)}} \frac{\partial h_{t_1-2}}{\partial h_{(t_1-3)}} \dots$$

$$\frac{\partial h_{t1}}{\partial h_{(t_1-1)}} = U$$

$$\frac{\partial h_{t1}}{\partial h_{(t_0+1)}} = U^{l_1-l_0+1}$$

What happens if we use a saturated non-linearity?

$$\frac{\partial h_{l1}}{\partial x_{t_0+1}} = \frac{\partial h_{t1}}{\partial h_{(t_0+1)}} \frac{\partial h_{t_0+1}}{\partial x_{(t_0+1)}}$$

$$\frac{\partial h_{t1}}{\partial h_{(t_0+1)}} = \frac{\partial h_{t1}}{\partial h_{(t_1-1)}} \frac{\partial h_{t_1-1}}{\partial h_{(t_1-2)}} \frac{\partial h_{t_1-2}}{\partial h_{(t_1-3)}} \dots$$

$$\frac{\partial h_{t1}}{\partial h_{(t_1-1)}} = U$$

$$\frac{\partial h_{t1}}{\partial h_{(t_0+1)}} = U^{l_1-l_0+1}$$

Exploding Gradients are easy to solve:

Clip!

$$\tilde{\nabla} = \begin{cases} \tau \frac{\nabla}{||\nabla||} & if\ ||\nabla|| > \tau \\ \nabla & \end{cases}$$

GRUs and LSTMs fix the vanishing gradient problem

- We had: $\frac{\partial h_{t1}}{\partial h_{(t_0+1)}} = \frac{\partial h_{t1}}{\partial h_{(t_1-1)}} \frac{\partial h_{t_1-1}}{\partial h_{(t_1-2)}} \frac{\partial h_{t_1-2}}{\partial h_{(t_1-3)}} \ldots$

- With the GRU, we have:

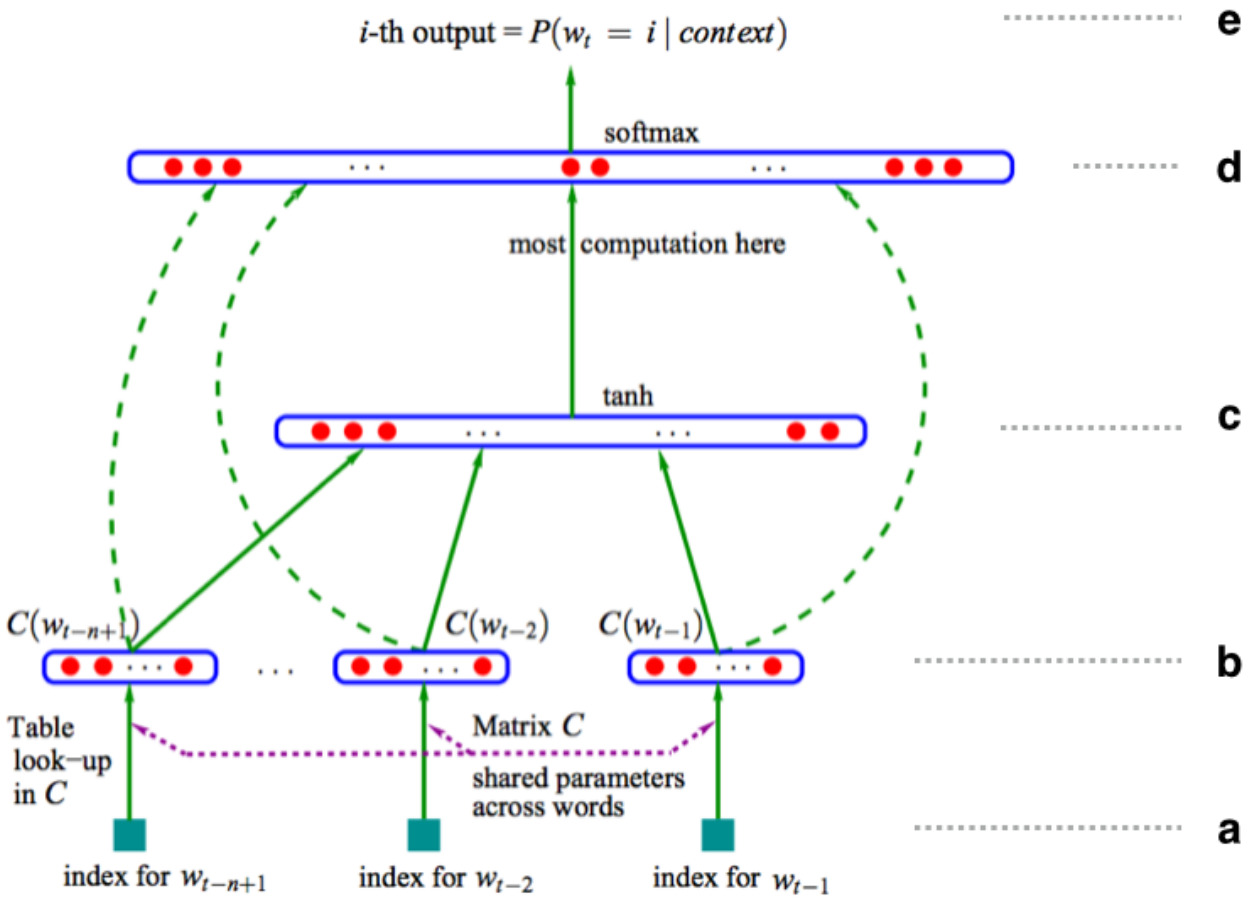
$$\frac{\partial h_t}{\partial h_{t-1}} = 1 - u + (u) \frac{\partial h'}{\partial h_{t-1}}$$

$$= 1 \; (if \; u = 0)$$

- Essentially a skip connection if the update gate is shut

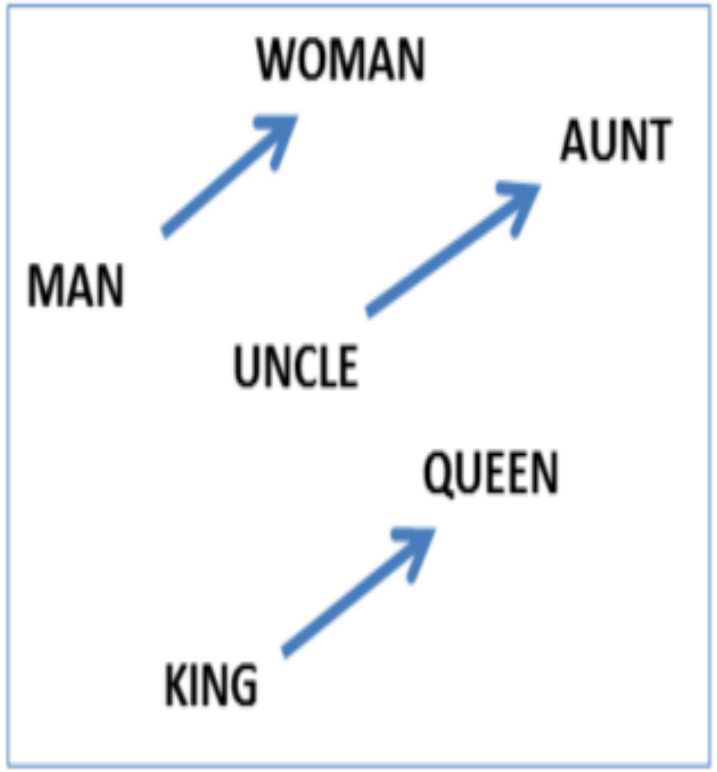
Neural Language Model

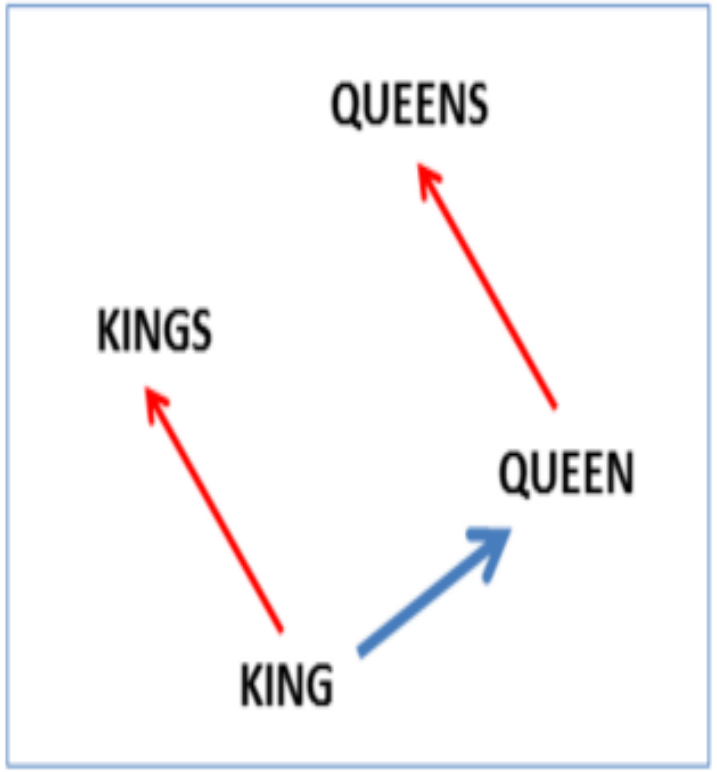
$$softmax(x) = \frac{e^{\beta x_j}}{\sum_i e^{\beta x_i}}$$



Source: [Bengio et al [2003]]

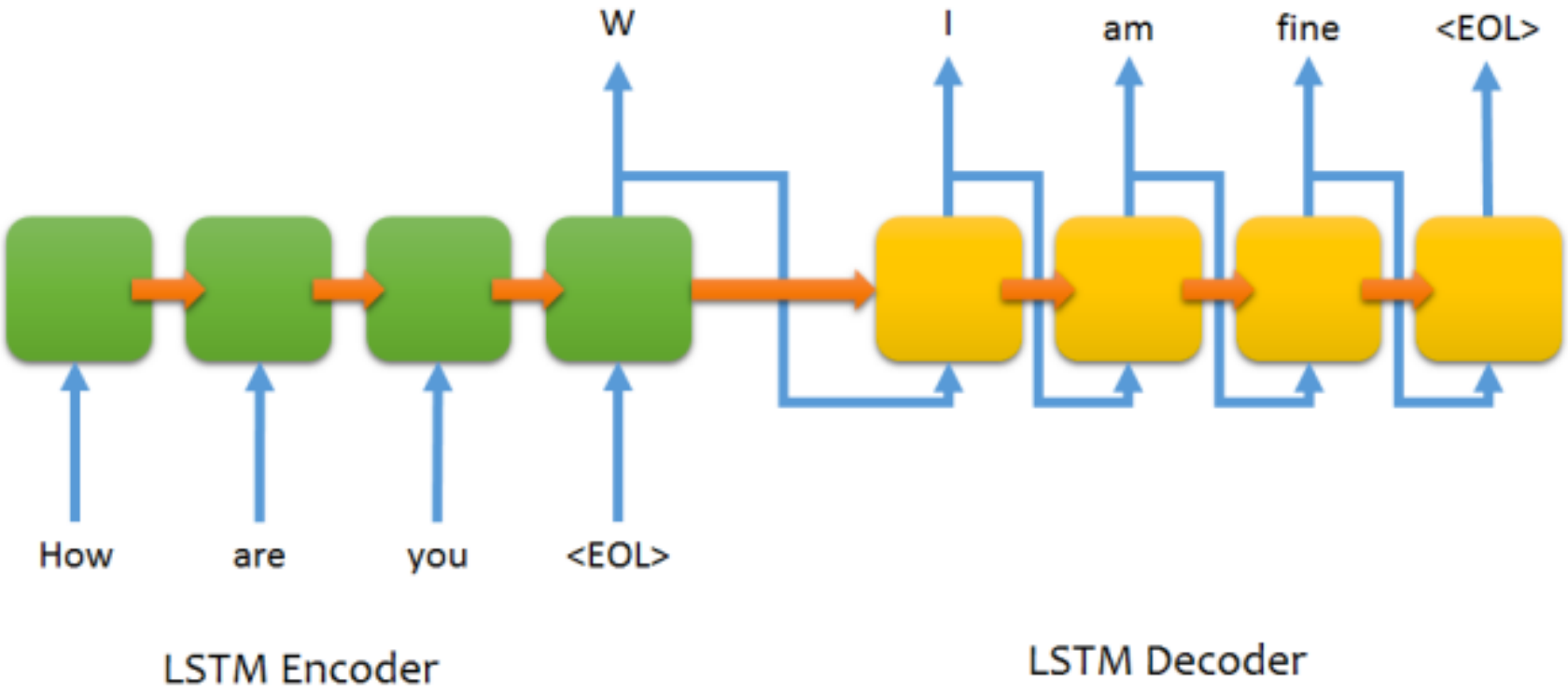
Are the word embeddings magic?





Mikolov et al shocked the world a little bit when the representations they learned

Seq2Seq Models

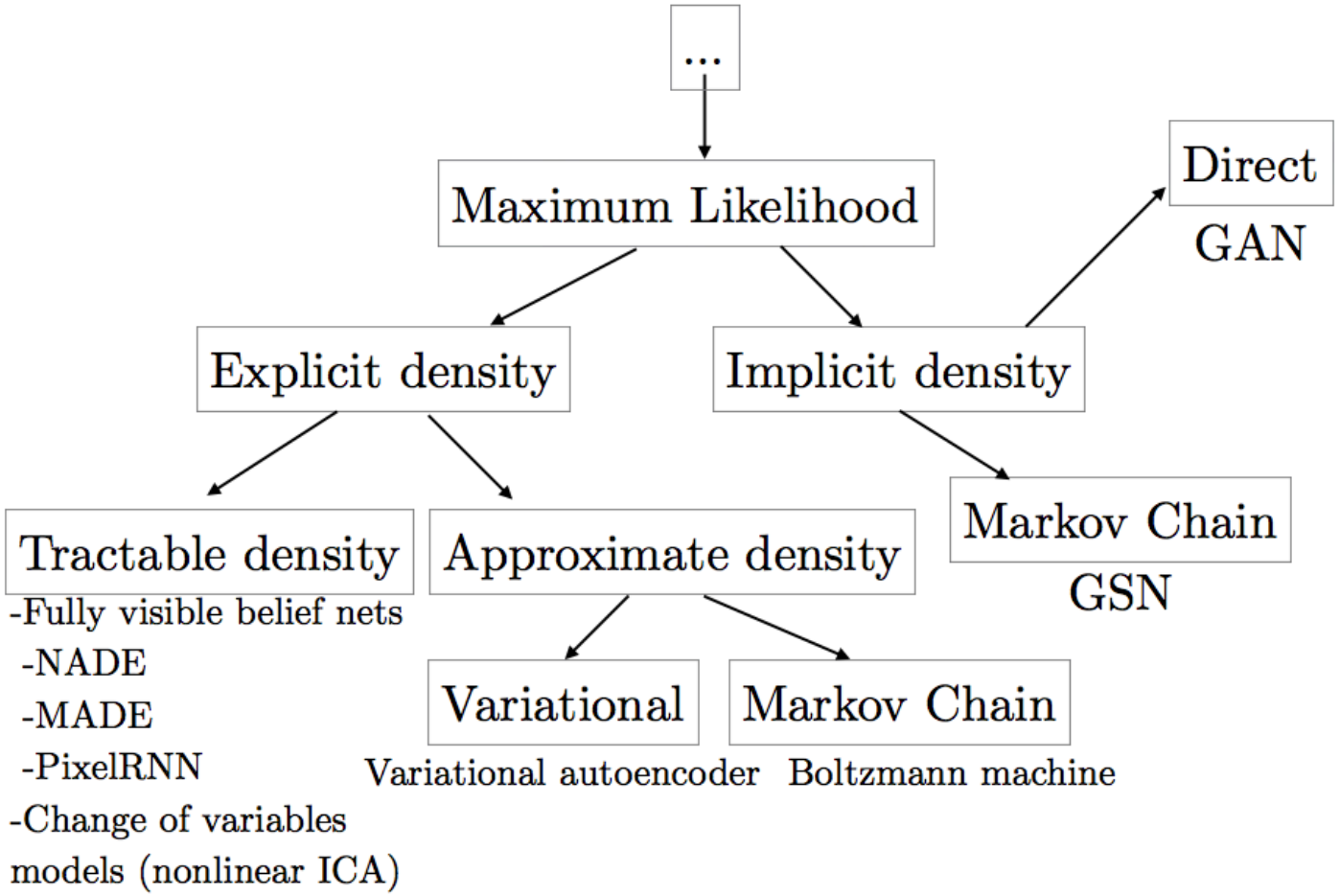


Source: [Sutskever et al [2014]]

Supervised vs Unsupervised Learning

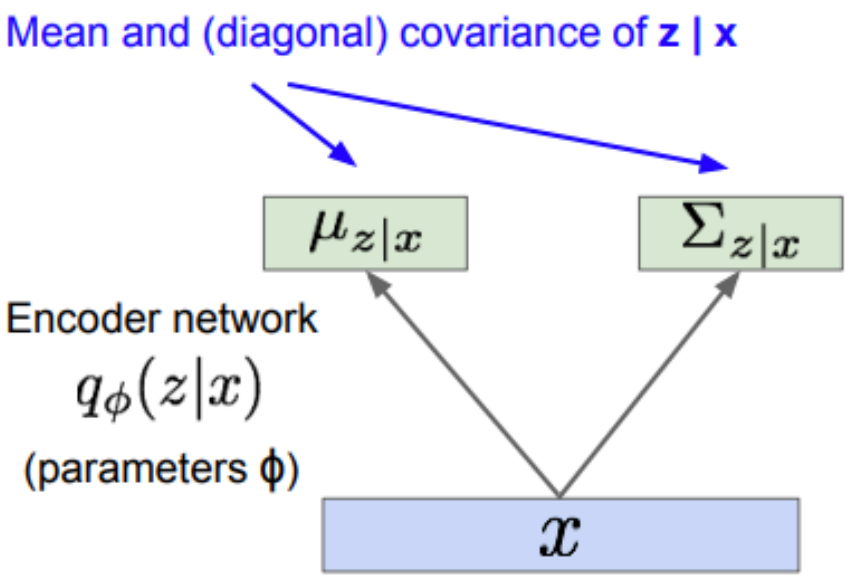
- Supervised Learning → Learn $P(Y|X)$
- Unsupervised Learning → Learn $P(X)$

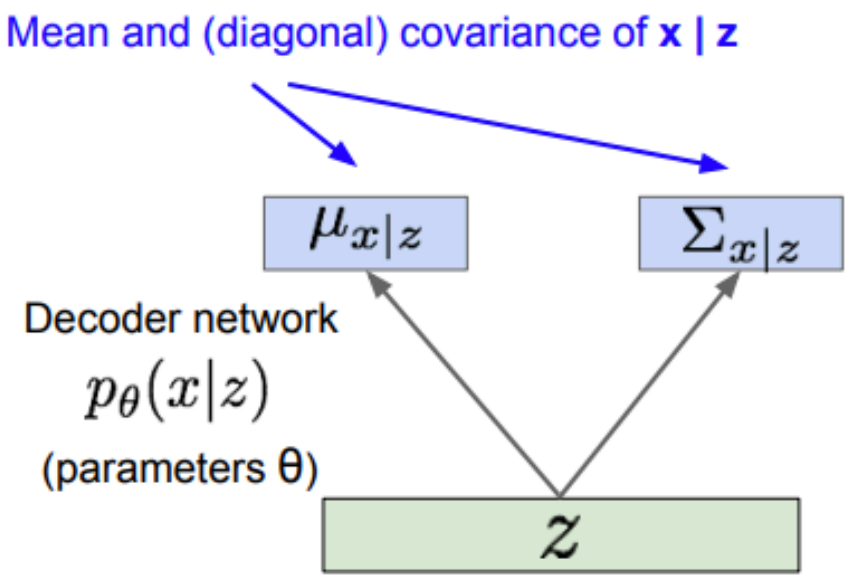
Generative Models



Source: [Goodfellow (2016)]

Variational Autoencoders





Make approximate posterior distribution close to prior

ELBO $$\log p_\theta(x_i) \geq \mathbb{E}[\log p_\theta(x_i|z)] - D_{KL}(q_\theta(z|x_i)||p_\theta(z))$$

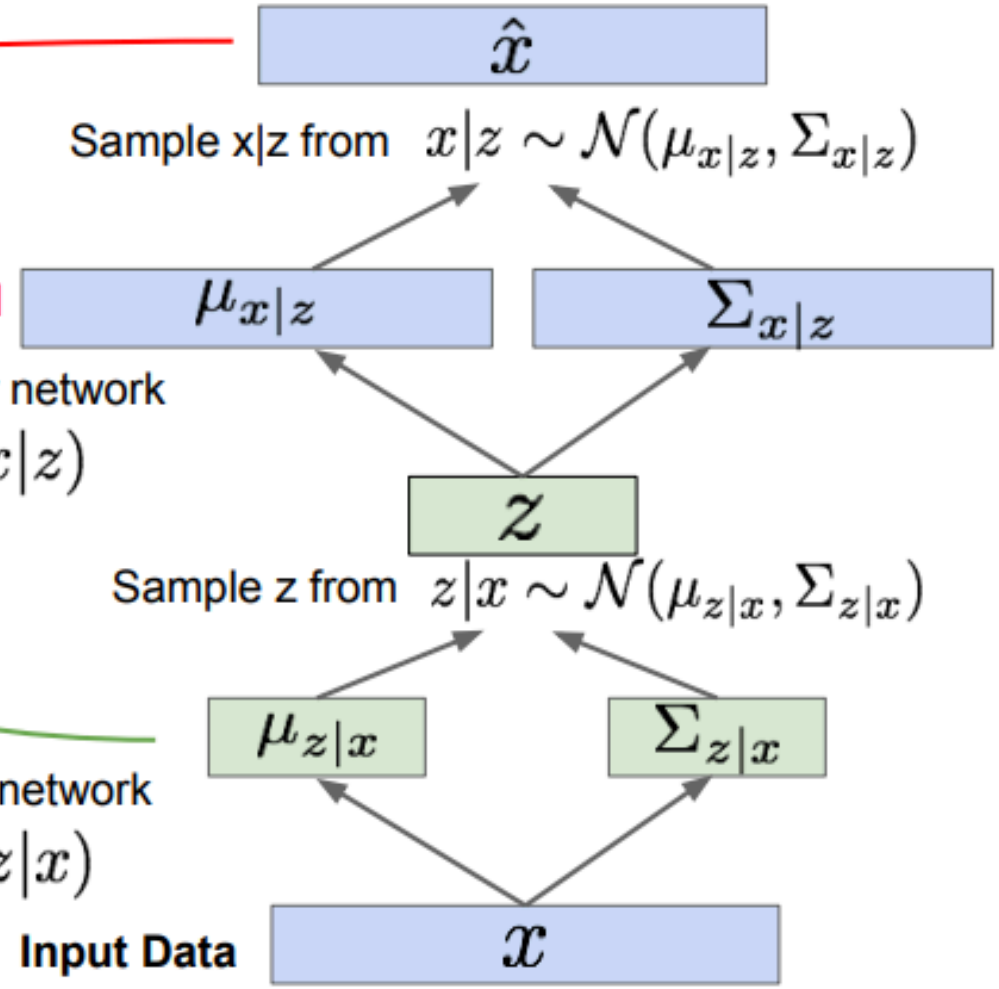
Reconstruct the input data

Variational Autoencoders

Putting it all together: maximizing the likelihood lower bound

$$\underbrace{\mathbf{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right] - D_{KL}(q_\phi(z \mid x^{(i)}) \,||\, p_\theta(z))}_{\mathcal{L}(x^{(i)},\theta,\phi)}$$

Maximize likelihood of original input being reconstructed

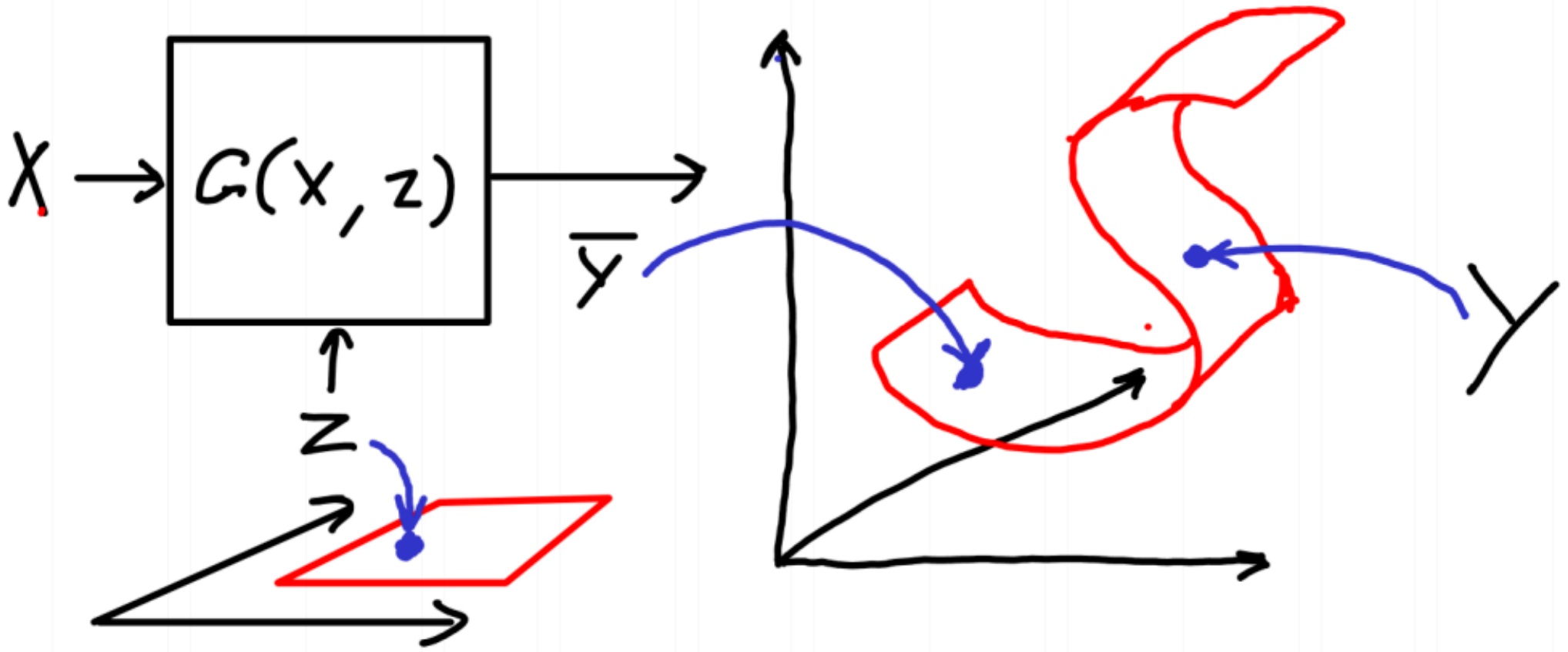


Make approximate posterior distribution close to prior

For every minibatch of input data: compute this forward pass, and then backprop!

Source: Karpathy (2015)

Generative Adversarial Networks



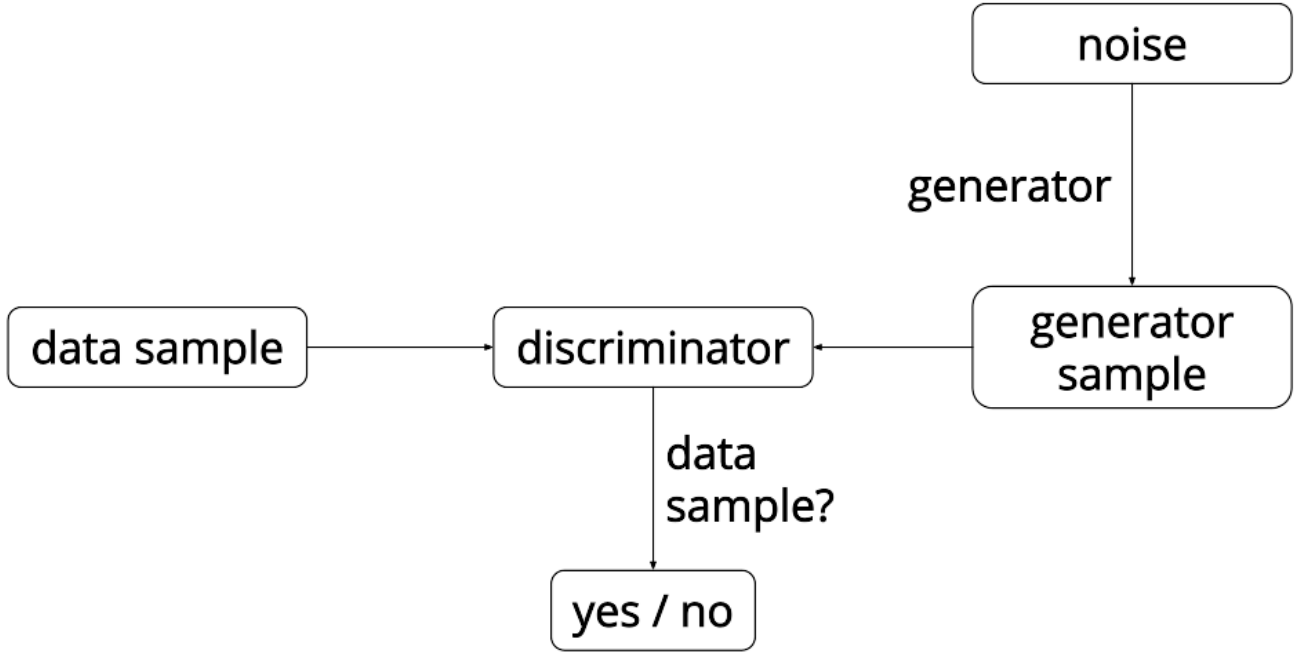
Source: [LeCun (2016)]

Generative Adversarial Networks

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

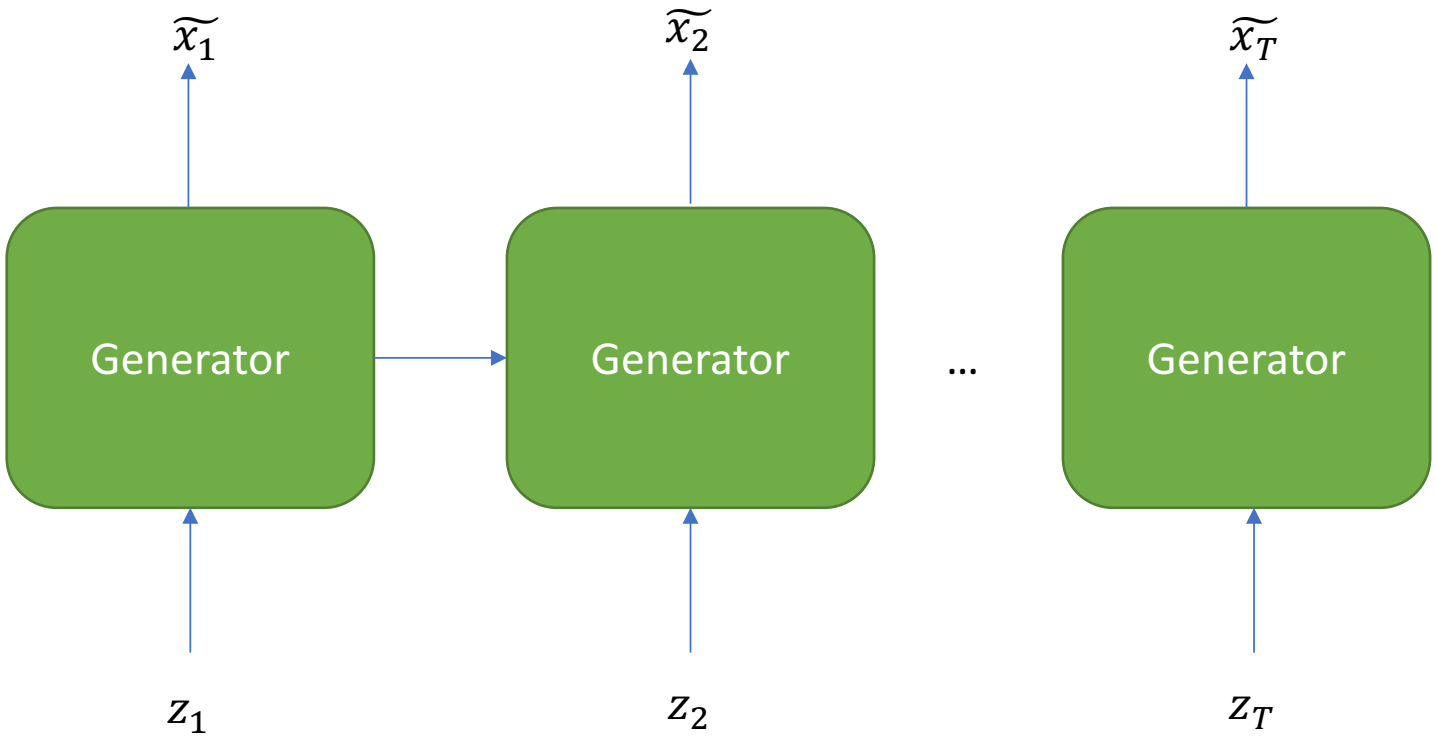
$$D_{G^*}(x) = \frac{p_{data}(x)}{p_{data}(x) + p_G(x)}$$

$$\Longrightarrow p_g(x) = p_{data}(x)$$



Let’s try a simple example

- Our data is a geometric brownian (log-normal) path
- The discriminator takes actual paths and the generator takes a single 100 dimensional vector of noise

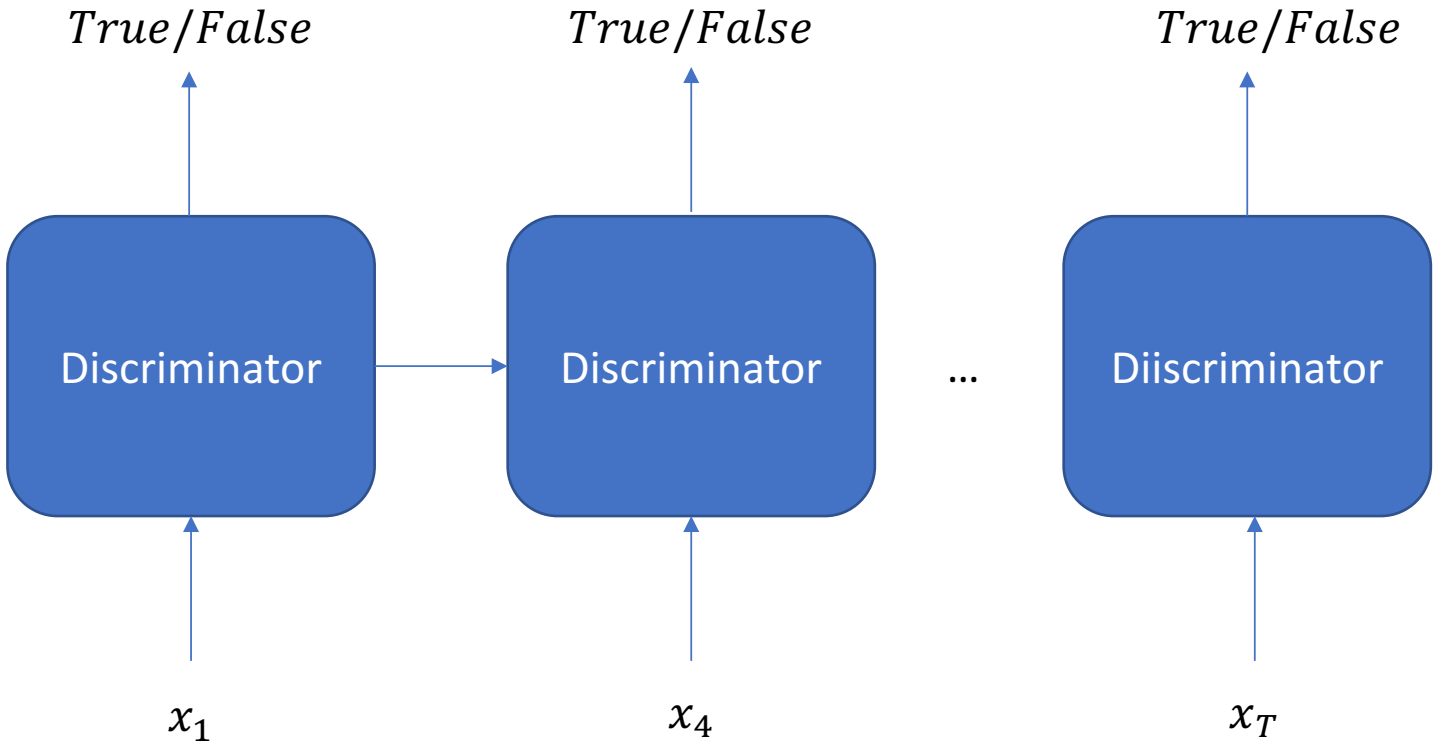


Let’s try a simple example

- Our data is a geometric brownian (log-normal) path
- The discriminator takes actual paths and the generator takes a single 100 dimensional vector of noise

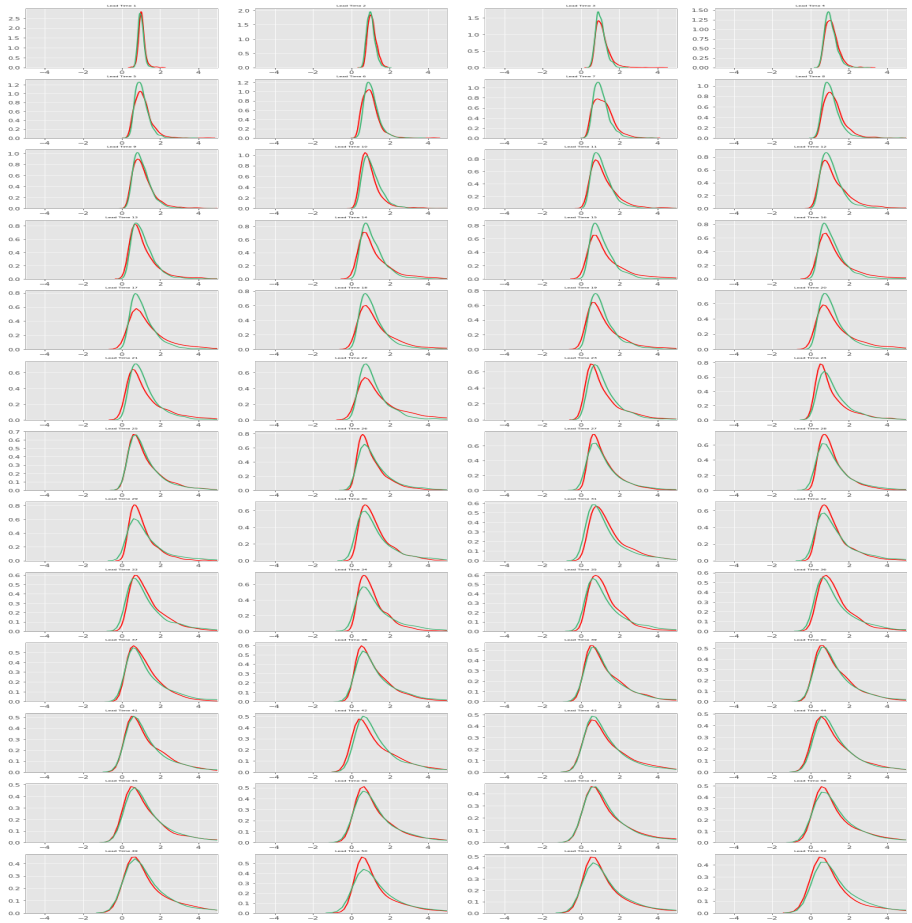
Discriminator can also be many to 1

For many to many – usually you use the median value of all the scores to classify the time series



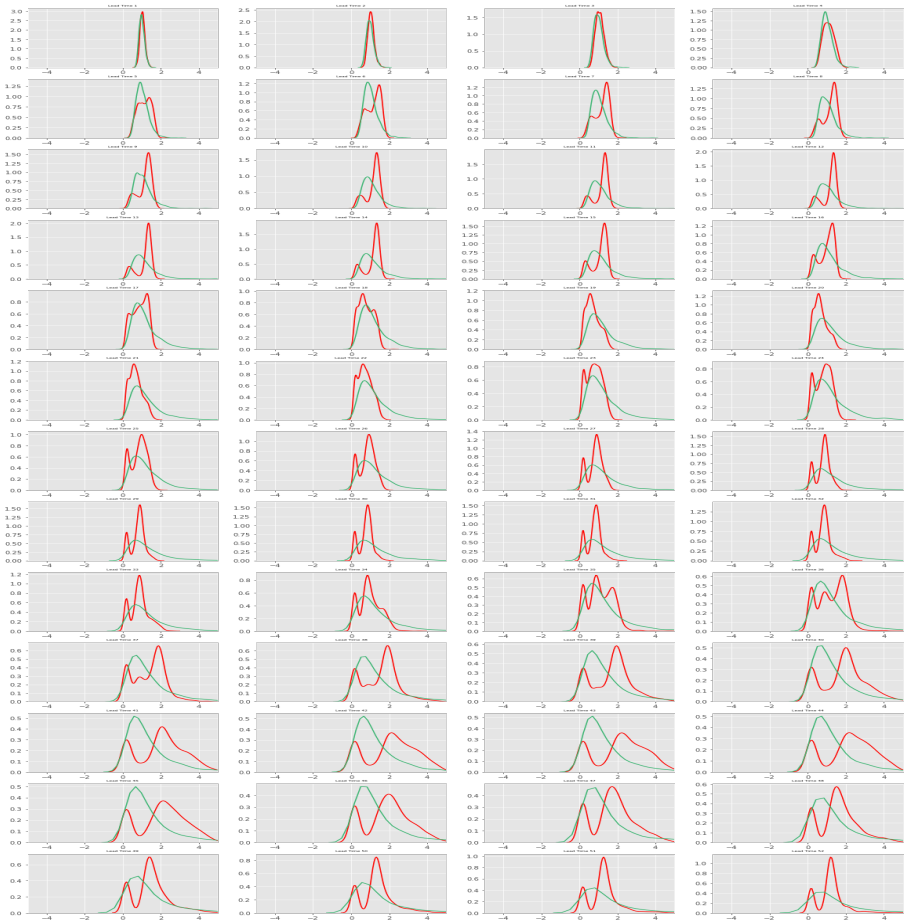
Does it work?

- Let’s try to learn a Brownian Motion

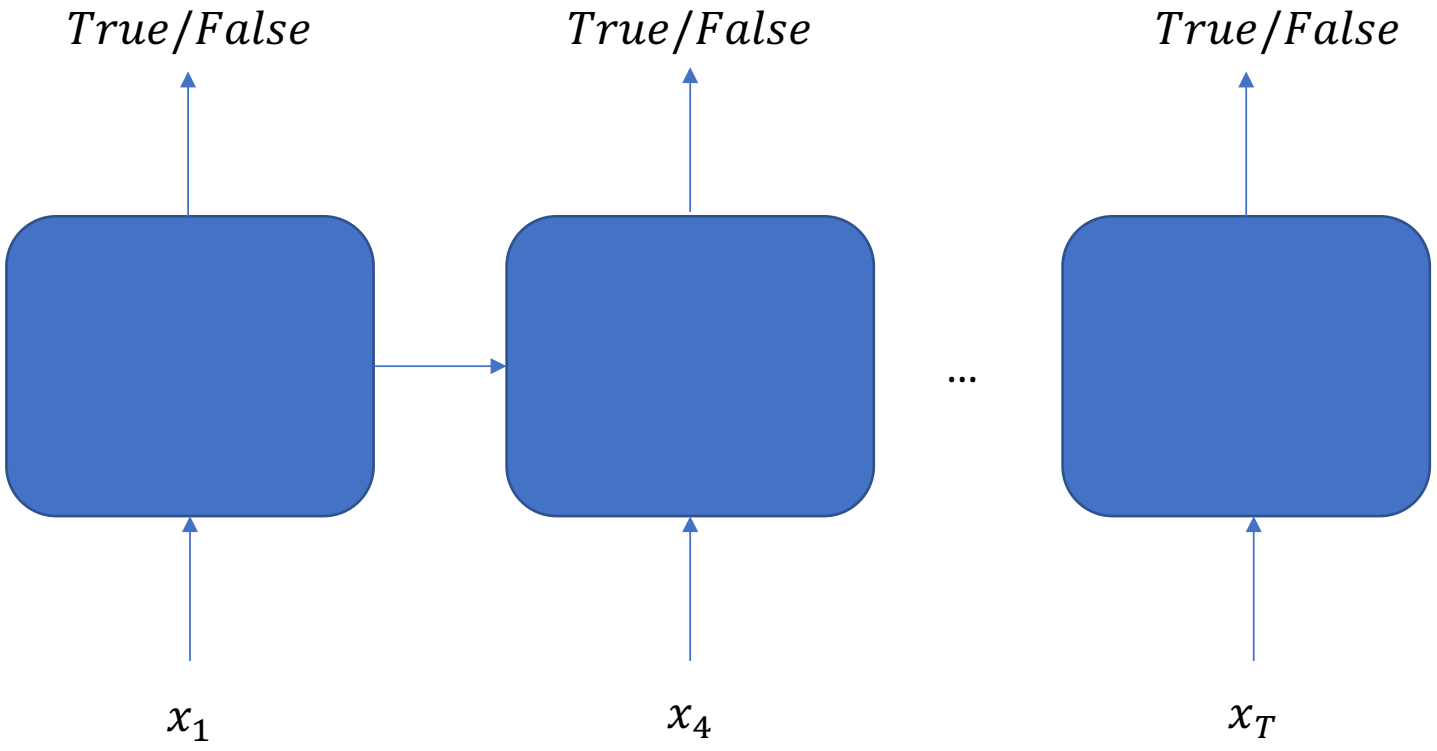


RNN’s tend to be very unstable

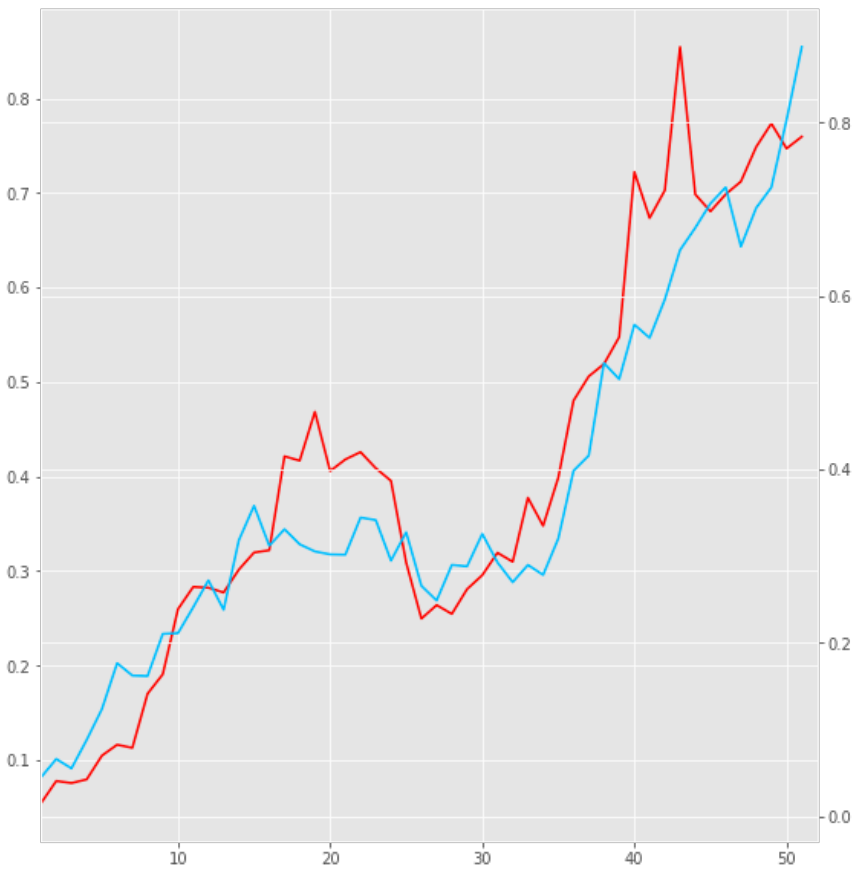
DCGAN on the other hand – works like a charm



Move Based RNN's



Move Based RNN's



Helping the discriminator focus on what's relevant

The move from JS to WGAN has the focus on making the discriminator Lipschitz.

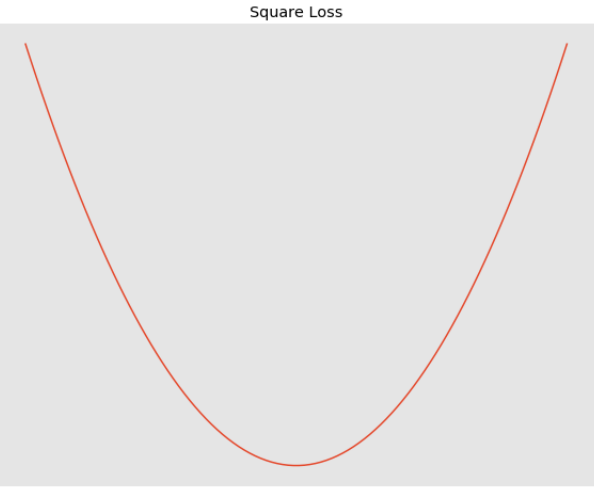
Spectral Normalization works for Convs/MLPs but for RNNs the notion of Lipschitz is not even clearly defined

A Financial Example

Can supervised learning learn an entire density?

- Sure! Make your loss function optimize for a quantile (or all of them)!

Little pictures



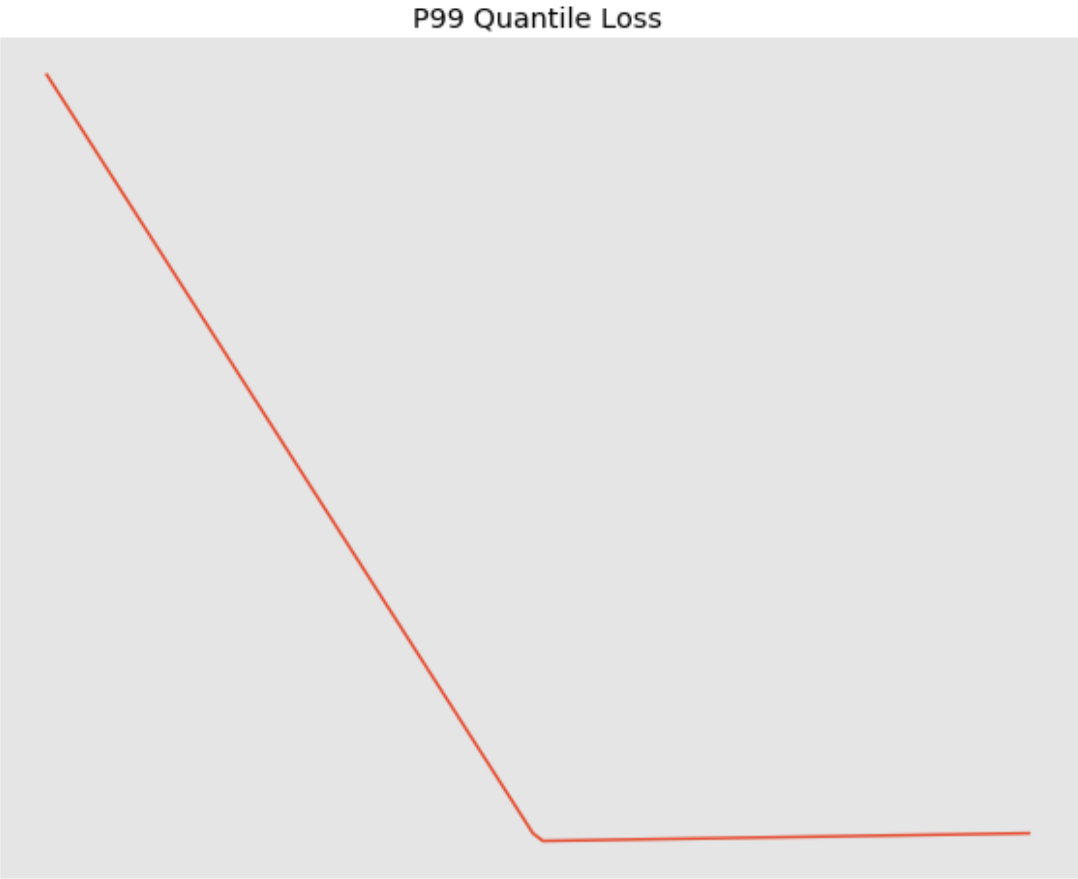
Mean

Median

P90 Quantile

Can supervised learning learn an entire density?

- For VaR use P99 quantile loss:



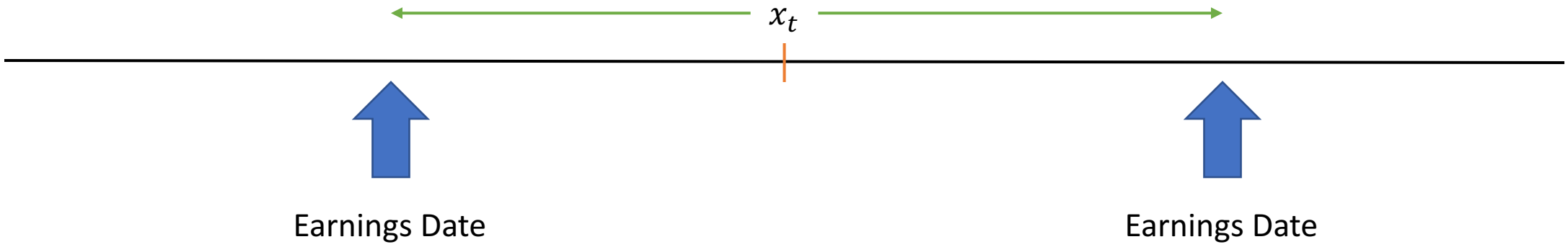
Penalize underpredictions 99 times more than overpredictions!

Can supervised learning learn an entire density?

- Quantile Loss: $QL_\alpha(x, y) = \alpha * (y - x)^+ + (1 - \alpha) * (x - y)^+$
- Penalize over or underpredictions more – depending on which quantile you'd like to predict!
- Better way to predict VaR – no distributional assumptions to show that P_α minimizes the quantile loss
- Works surprisingly well with NN's/RNN's – if you predict each T_1/T_2 individually! (Wen et al NIPS 2017)

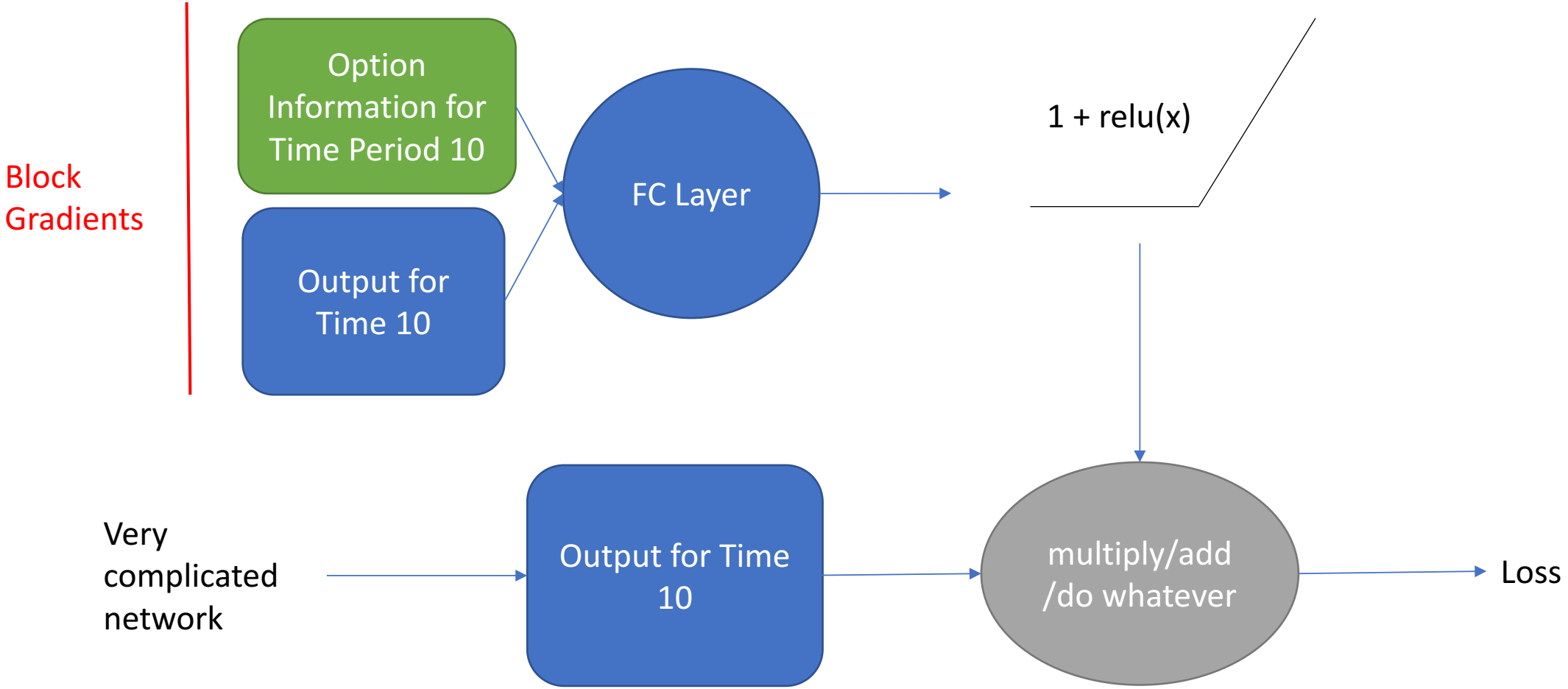
Don’t forget to teach your network to look for additional volatility

- Distance to earnings:



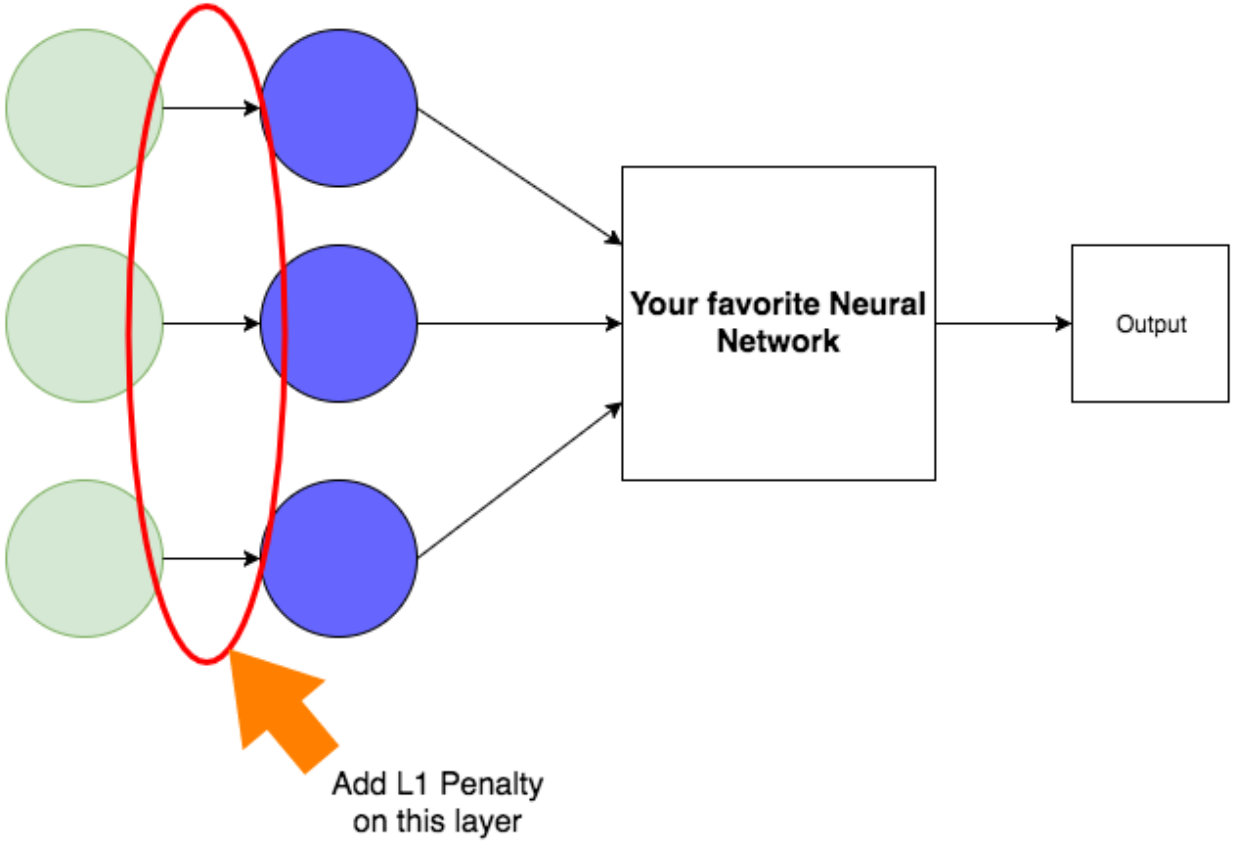
Finally, make sure you add Future Information

- Targeted input alignment:

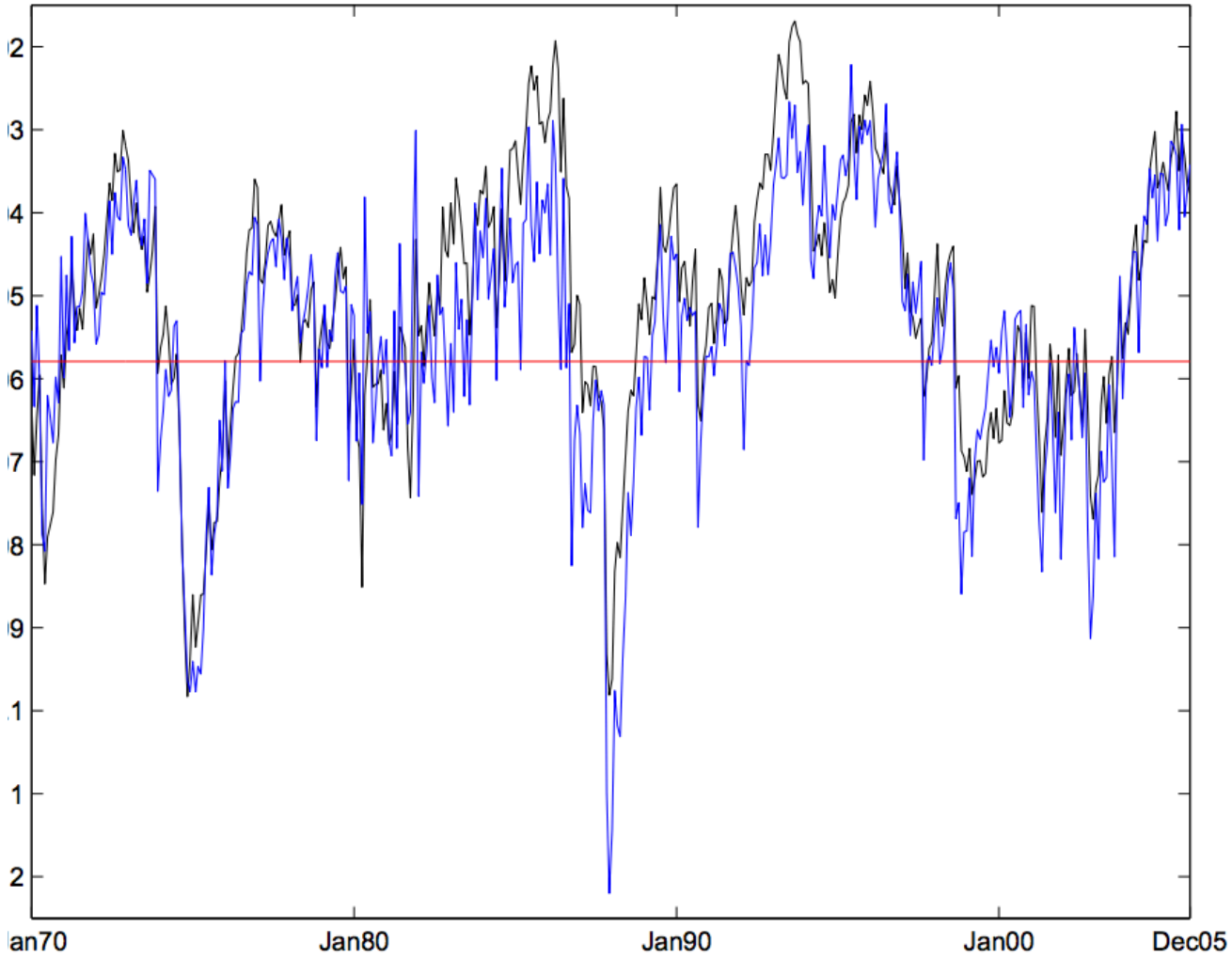


Architecture

- MLP(SNN! – Typically, 2-4 Epoch convergence)
- ConvLSTM/GRU
- Dropout to prevent overfitting in RNN's
- If you want to throw the kitchen sink of data, make sure to do Feature Selection (correctly):

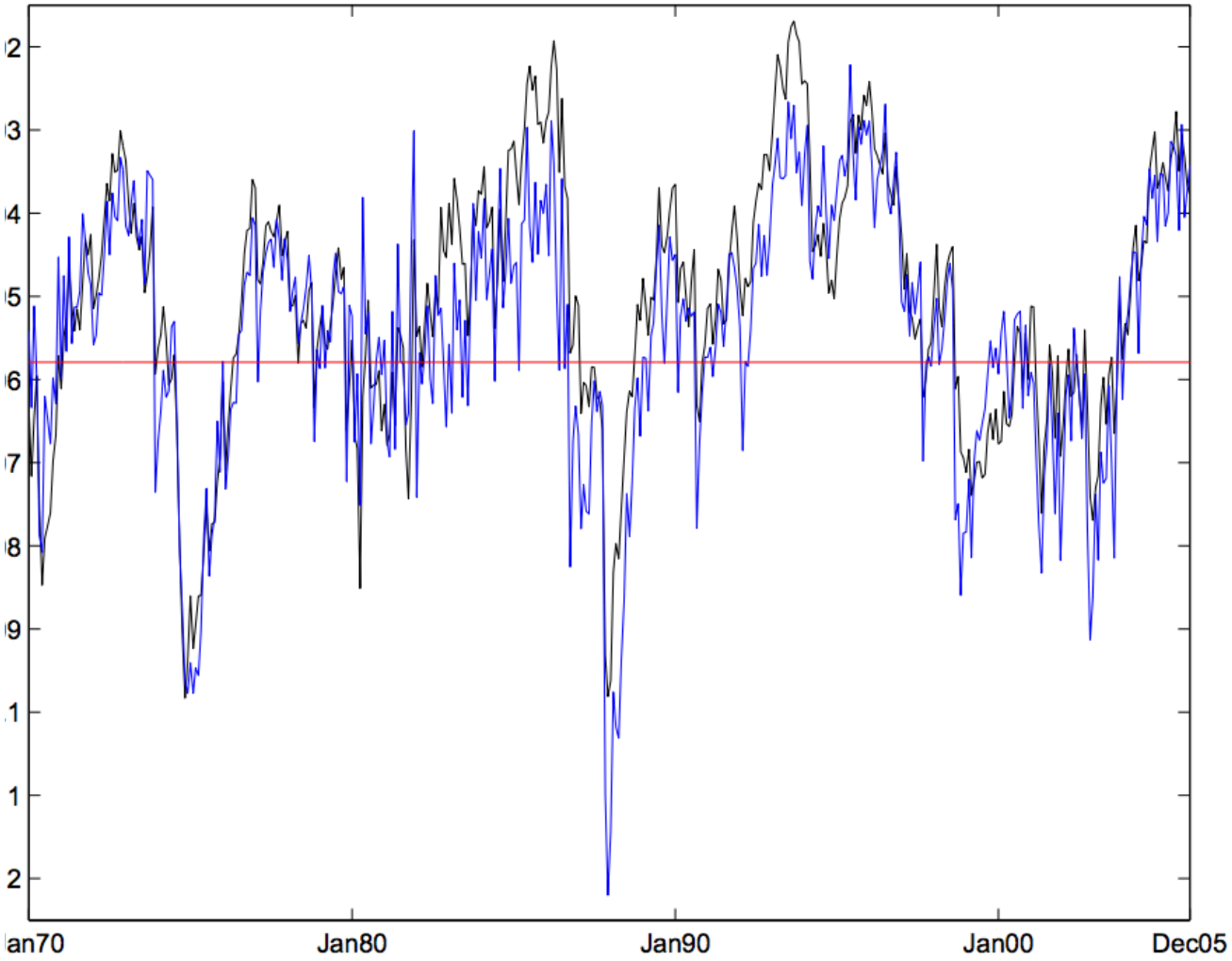


Does it work?

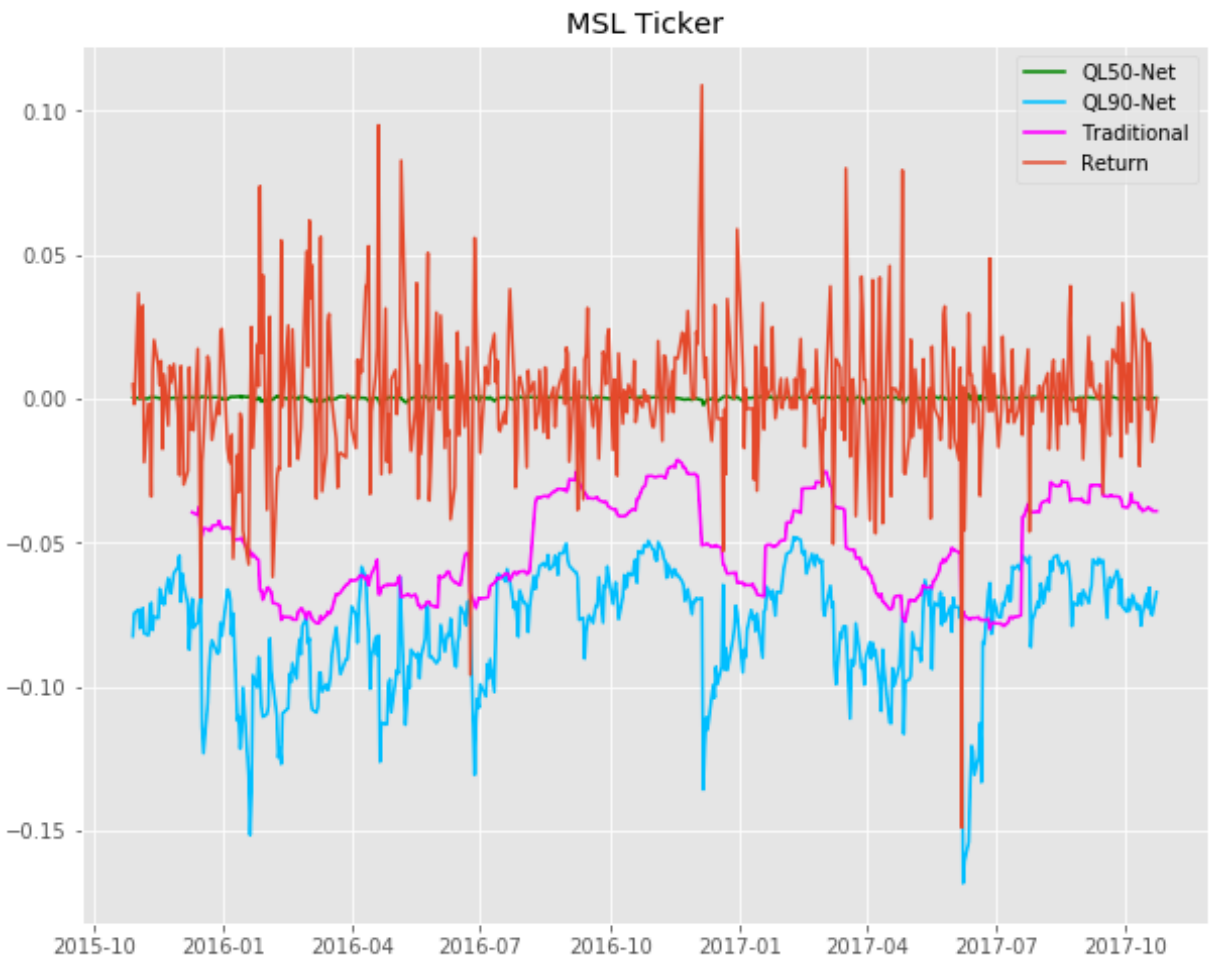


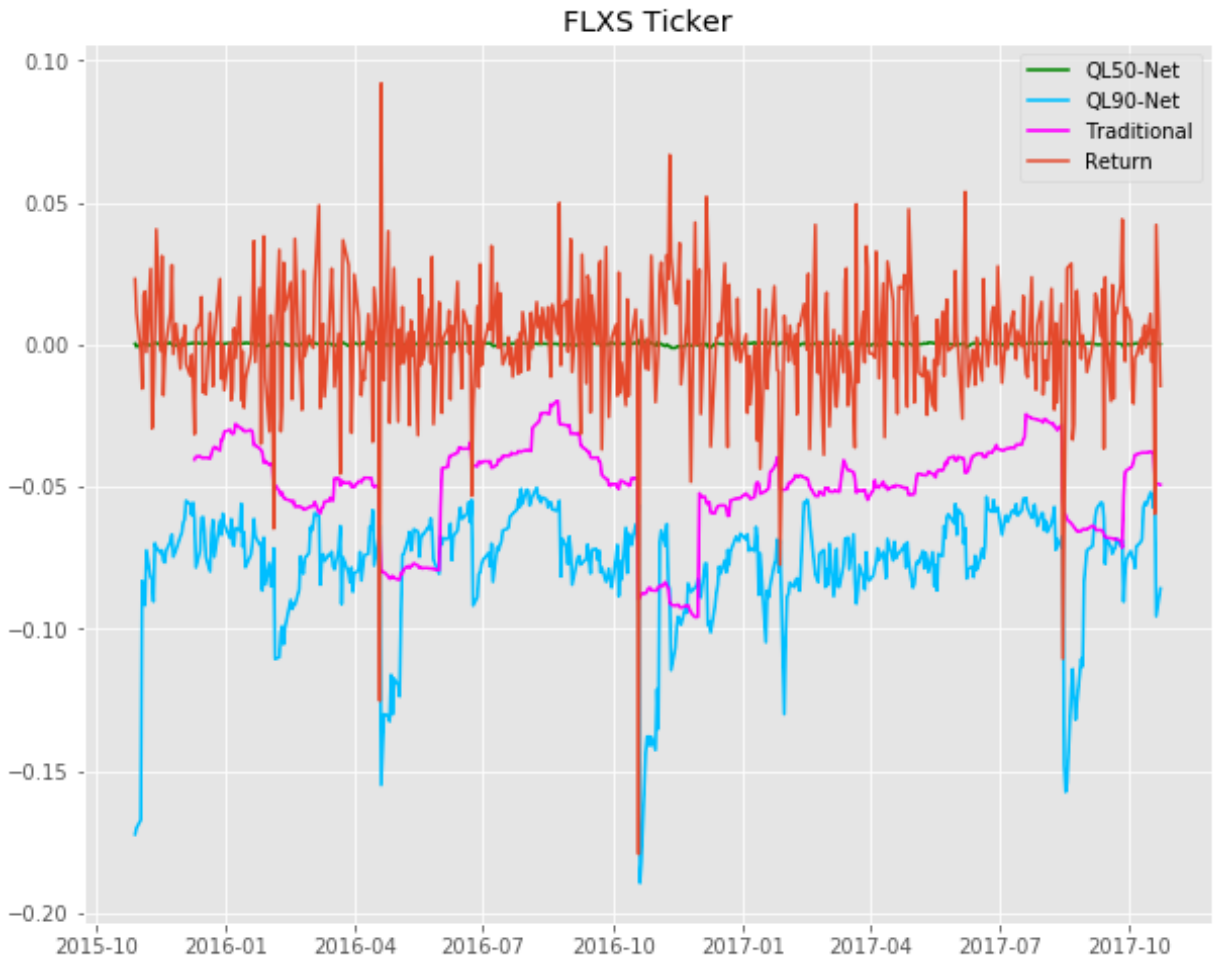
Does it work?

Cenesizoglu et al 2007

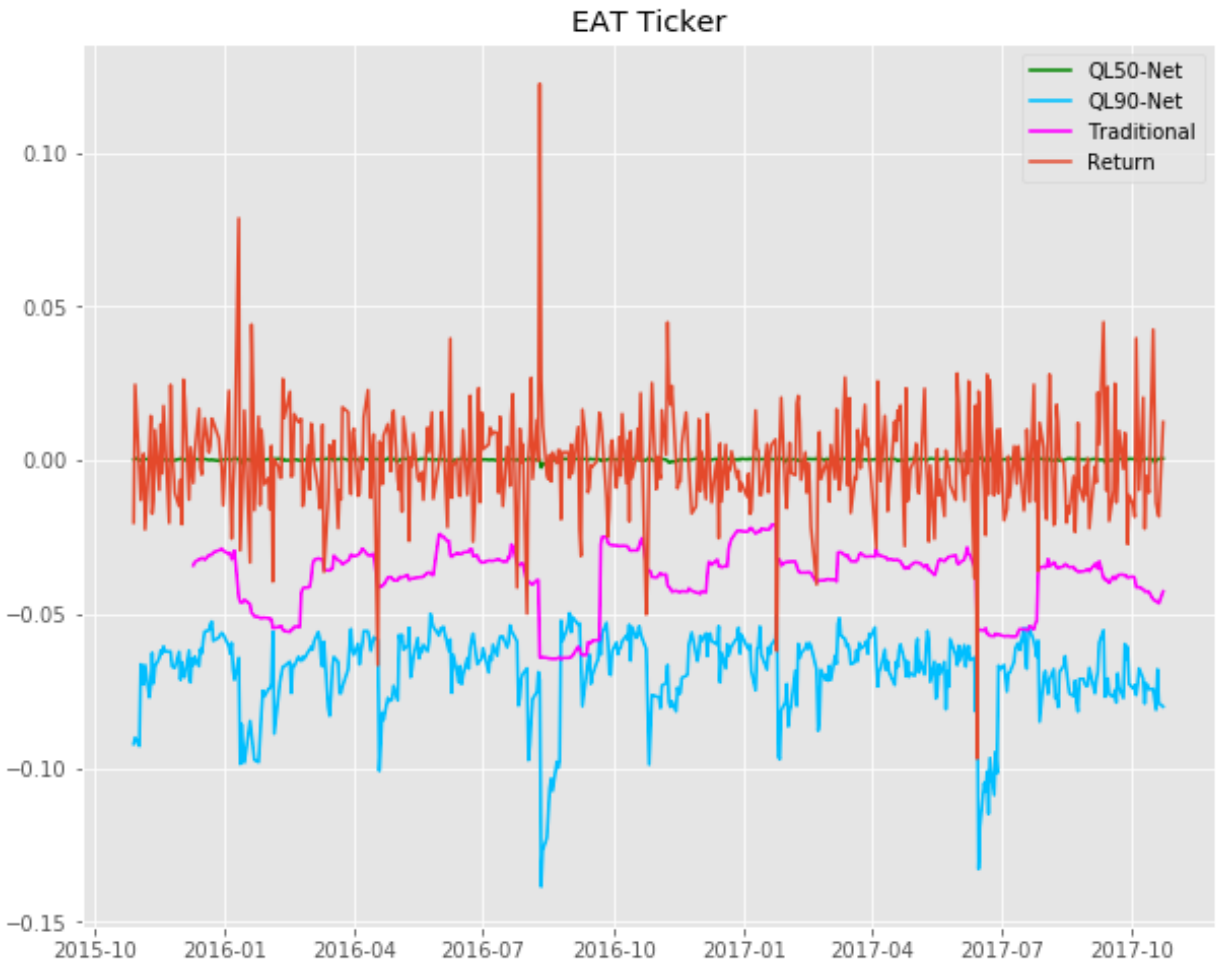


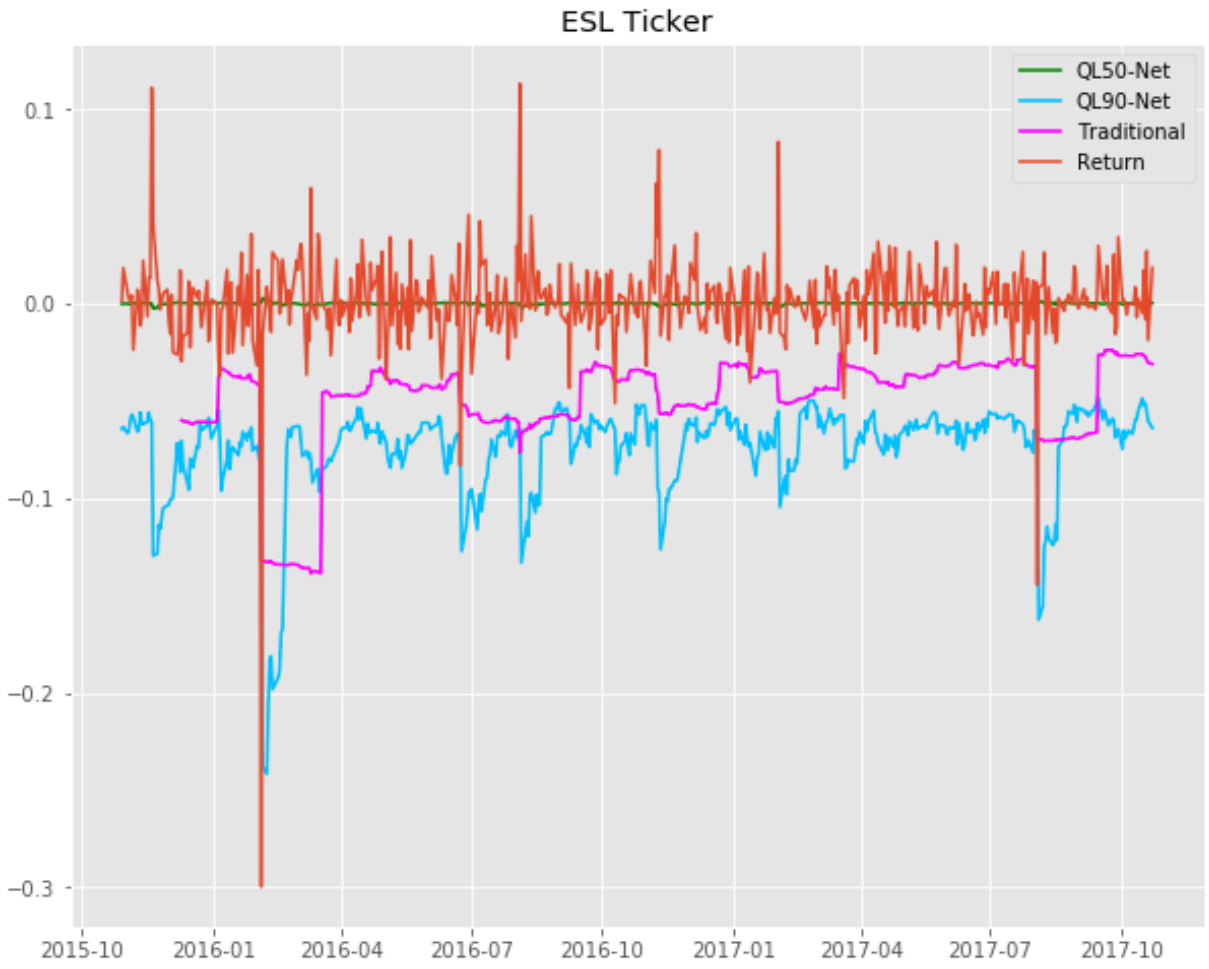
Does it work?


MSL Ticker
QL50-Net
QL90-Net
Traditional
Return
0.10
0.05
0.00
−0.05
−0.10
−0.15
2015-10
2016-01
2016-04
2016-07
2016-10
2017-01
2017-04
2017-07
2017-10

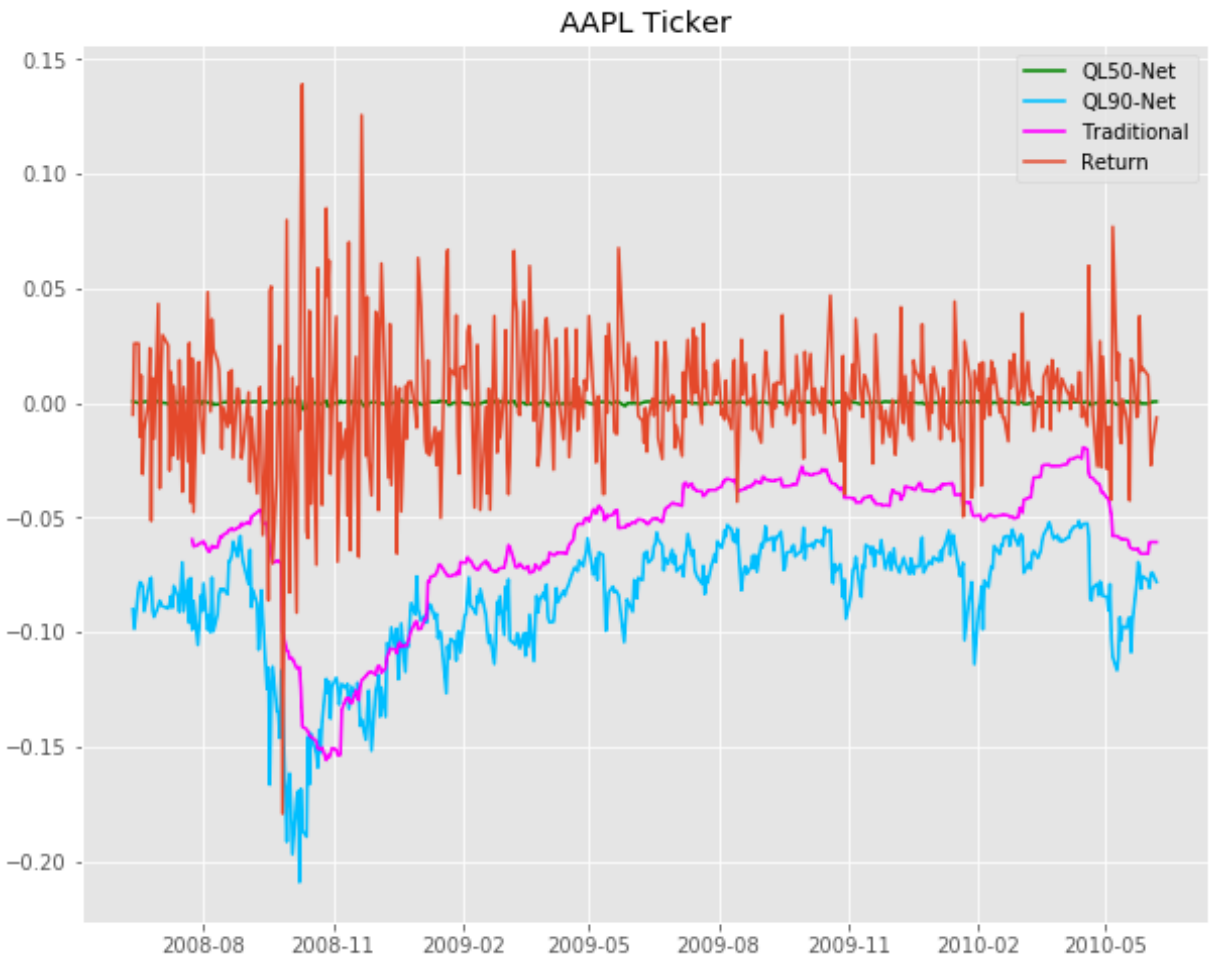

FLXS Ticker
QL50-Net
QL90-Net
Traditional
Return
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
2015-10
2016-01
2016-04
2016-07
2016-10
2017-01
2017-04
2017-07
2017-10

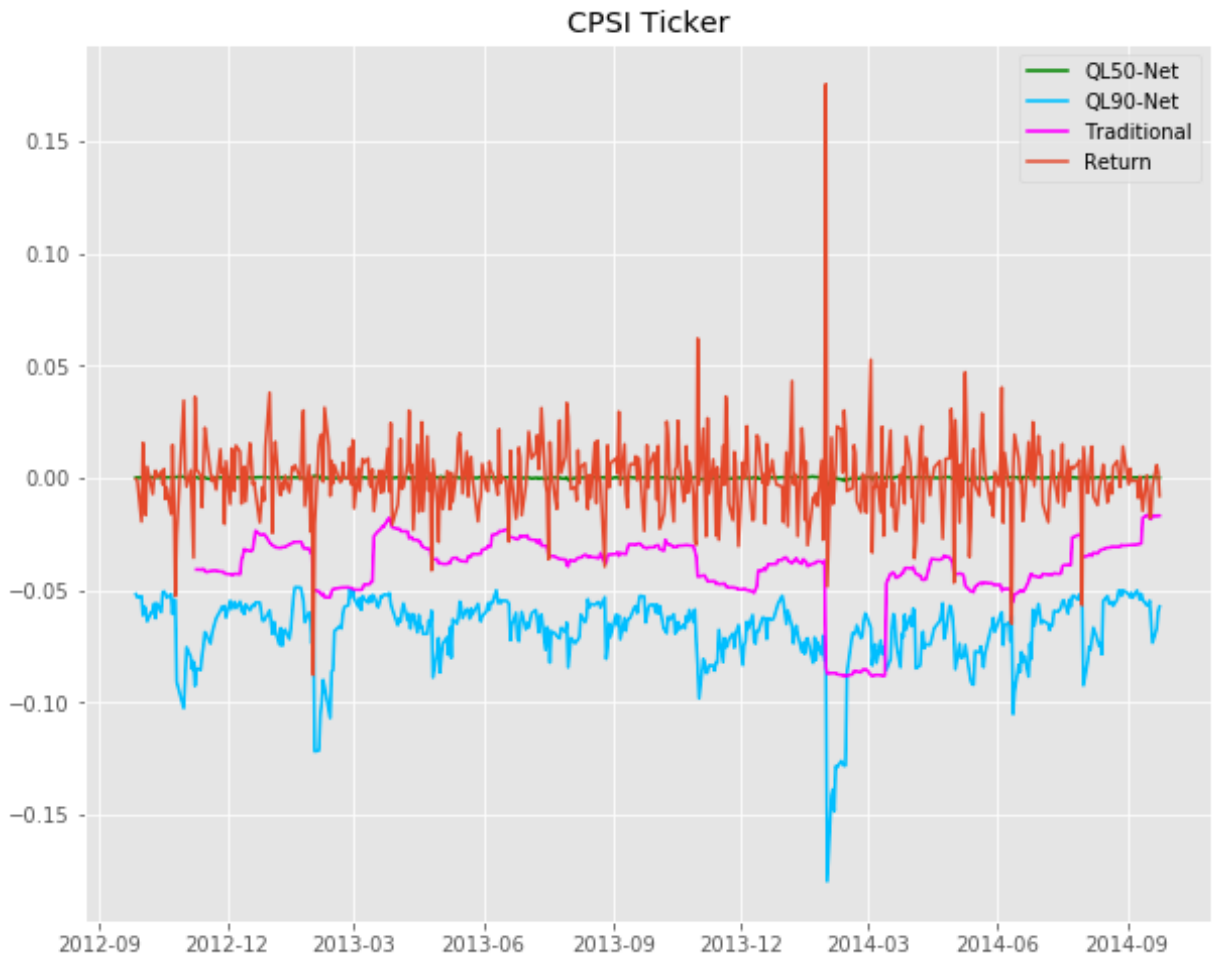
Does it work?


EAT Ticker
QL50-Net
QL90-Net
Traditional
Return
0.10
0.05
0.00
−0.05
−0.10
−0.15
2015-10
2016-01
2016-04
2016-07
2016-10
2017-01
2017-04
2017-07
2017-10


ESL Ticker
QL50-Net
QL90-Net
Traditional
Return
0.1
0.0
−0.1
−0.2
−0.3
2015-10
2016-01
2016-04
2016-07
2016-10
2017-01
2017-04
2017-07
2017-10

Does it work?


AAPL Ticker
QL50-Net
QL90-Net
Traditional
Return
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
2008-08
2008-11
2009-02
2009-05
2009-08
2009-11
2010-02
2010-05


CPSI Ticker
QL50-Net
QL90-Net
Traditional
Return
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
2012-09
2012-12
2013-03
2013-06
2013-09
2013-12
2014-03
2014-06
2014-09

Conclusion

- That was a lot
- Deep Methods are very promising
- Building constraints is not that hard – you just need a good regularizer
- Things I didn’t cover which are super cool:
  - Localization/Detection
  - Attention
  - Recommender Systems
  - Deep Reinforcement Learning
  - 1000 other things