Best choice of pivot item = median.

Sorting summary

	inplace?	stable?	worst	average	best	remarks
selection	✔		$N^2 / 2$	$N^2 / 2$	$N^2 / 2$	*N* exchanges
insertion	✔	✔	$N^2 / 2$	$N^2 / 4$	N	use for small *N* or partially ordered
shell	✔		?	?	N	tight code, subquadratic
merge		✔	N lg N	N lg N	N lg N	*N* log *N* guarantee, stable
quick	✔		$N^2 / 2$	2 N ln N	N lg N	*N* log *N* probabilistic guarantee fastest in practice
3-way quick	✔		$N^2 / 2$	2 N ln N	N	improves quicksort in presence of duplicate keys
???	✔	✔	N lg N	N lg N	N	holy sorting grail

1v1 CONV with 32 filters preserves spatial dimensions, reduces depth.

Data Augmentation

- Horizontal or vertical Flips.
- Random crops and scales.
- Color Jitter.
- Random mix/combinations of :
  - Translation
  - Rotation
  - Stretching
  - Shearing
  - lens distortions

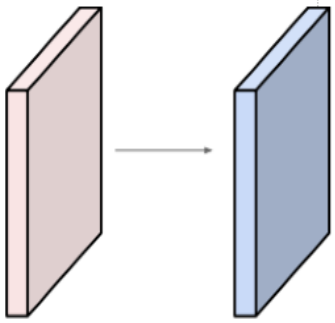
Transfer Learning

	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	You're in trouble... Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

Why CNN?

- local connectivity
- parameter sharing
- pooling / subsampling hidden units

Examples time:

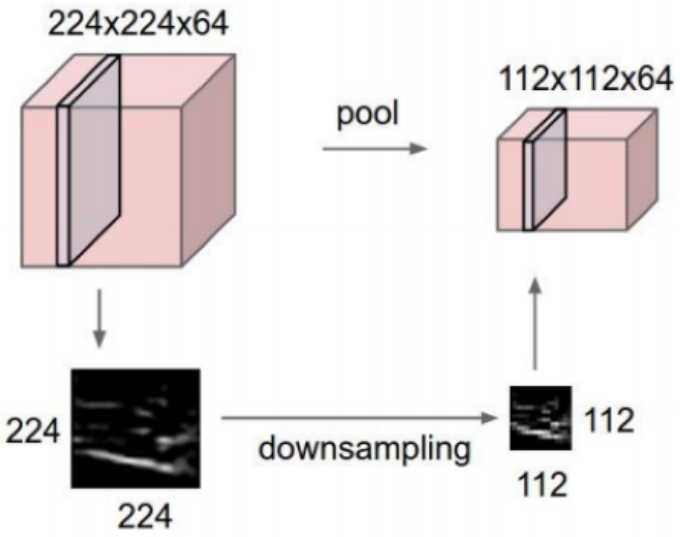


Input volume: **32x32x3**
10 5x5 filters with stride 1, pad 2

Number of parameters in this layer?
each filter has 5*5*3 + 1 = 76 params (+1 for bias)
=> 76*10 = **760**

Pooling layer

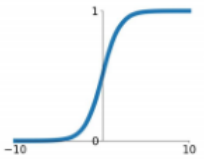
- Makes the representations smaller and more manageable.
- Operates over each activation map independently:



Activation Functions

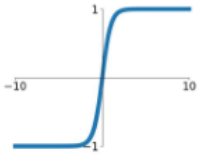
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



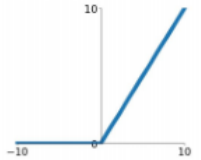
tanh

$\tanh(x)$



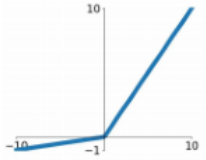
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

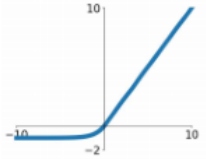


Maxout

$\max(w_1^T x + b_1, w_2^T x + b_2)$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Batch Normalization

[Ioffe and Szegedy, 2015]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Why RNN?

- Temporal dependencies
- Variable sequence length

Keras Simple RNN

```
inp = Input(shape = (10,8))
x = SimpleRNN(4)(inp)
model = Model(inp, x)
```

```
model.summary()
```

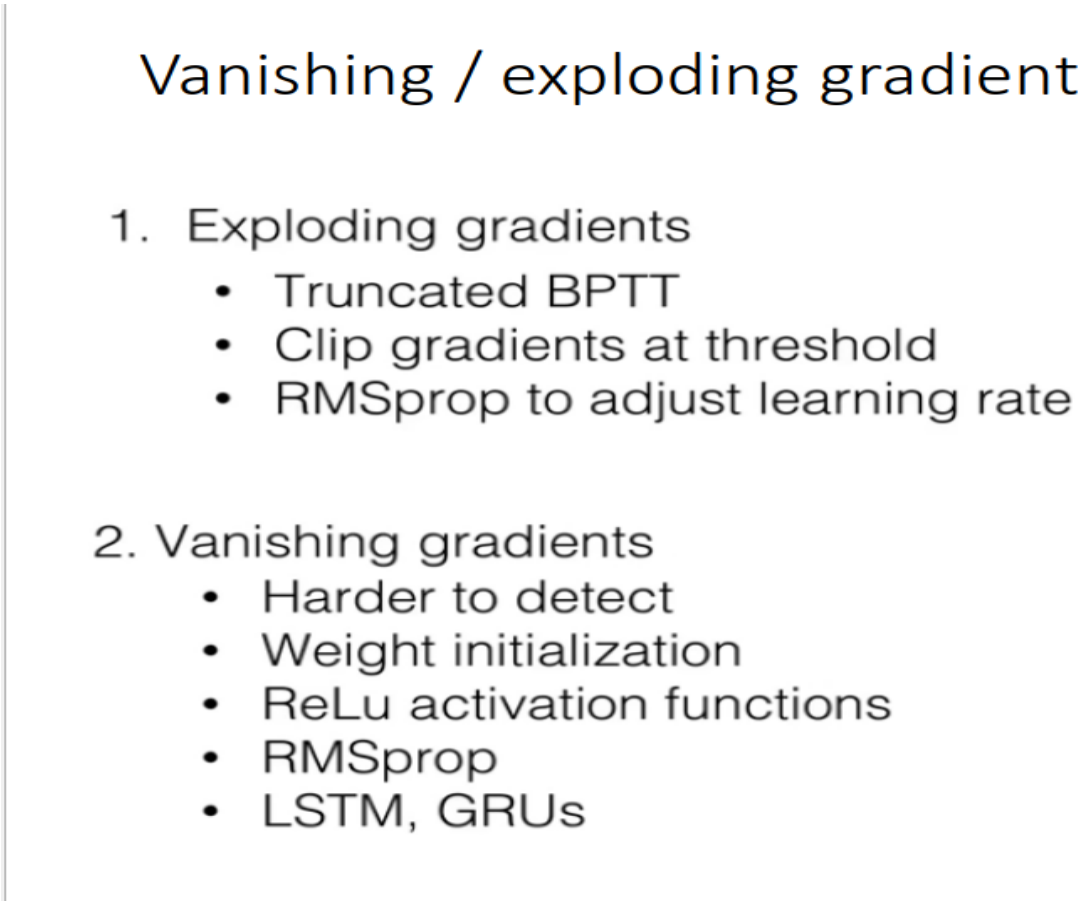
```
Model: "model_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_5 (InputLayer)        [(None, 10, 8)]           0

 simple_rnn_3 (SimpleRNN)    (None, 4)                 52

=================================================================
Total params: 52
Trainable params: 52
Non-trainable params: 0
_________________________________________________________________
```

Vanishing and Exploding gradient solutions

- tanh(a)
- max(a, 0)



LSTM (Long term short memory) networks

- They are capable of learning long-term dependencies.
  - LSTMs are explicitly designed to avoid the long-term dependency problem.
  - Remembering information for long periods of time is practically their default behavior, not something they struggle to learn.

Keras LSTM

```
inp = Input(shape=(28,28))
x = LSTM(16)(inp)
x = Dense(10,activation='softmax')(x)
model = Model(inp, x)
model.summary()
```

```
Model: "model_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_9 (InputLayer)        [(None, 28, 28)]          0

 lstm (LSTM)                 (None, 16)                2880

 dense_2 (Dense)             (None, 10)                170

=================================================================
Total params: 3,050
Trainable params: 3,050
Non-trainable params: 0
_________________________________________________________________
```

```
28*16*4+16*4+16*16*4
```

```
2880
```

Gated Recurrent Units (GRU)

- As RNNs and particularly the LSTM architecture rapidly gained popularity during the 2010s, a number of papers began to experiment with simplified architectures in hopes of retaining the key idea of incorporating an internal state and multiplicative gating mechanisms but with the aim of speeding up computation.
- The gated recurrent unit (GRU) (Cho et al., 2014) offered a streamlined version of the LSTM memory cell that often achieves comparable performance but with the advantage of being faster to compute (Chung et al., 2014).

Keras GRU

```
inp = Input(shape=(28,28))
x = GRU(16,reset_after=False)(inp)
x = Dense(10,activation='softmax')(x)
model = Model(inp, x)
model.summary()
```

```
Model: "model_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_12 (InputLayer)       [(None, 28, 28)]          0

 gru_2 (GRU)                 (None, 16)                2160

 dense_5 (Dense)             (None, 10)                170

=================================================================
Total params: 2,330
Trainable params: 2,330
Non-trainable params: 0
_________________________________________________________________
```

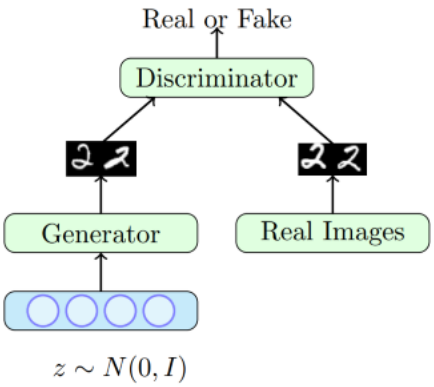
```
28*16*3+16*16*3+16*3
```

```
2160
```

Properties of Autoencoders

- **Data-specific**: Autoencoders are only able to compress data similar to what they have been trained on.
- **Lossy:** The decompressed outputs will be degraded compared to the original inputs.
- **Learned automatically from examples:** It is easy to train specialized instances of the algorithm that will perform well on a specific type of input.

The intuition



- What can we use for such a complex transformation?
  A Neural Network
- How do you train such a neural network?
  Using a two player game
- There are two players in the game: a generator and a discriminator
- The job of the generator is to produce images which look so natural that the discriminator thinks that the images came from the real data distribution
- The job of the discriminator is to get better and better at distinguishing between true images and generated (fake) images

VAE Encoder

- The encoder takes input and returns parameters for a probability density (e.g., $q_\theta(z \mid x)$n): I.e., gives the mean and co-variance matrix.
- We can sample from this distribution to get random values of the lower-dimensional representation z.
- Implemented via a neural network: each input x gives a vector mean and diagonal covariance matrix that determine the Gaussian density
- Parameters θ for the NN need to be learned – need to set up a loss function.

- The discriminator wants to maximize the second term whereas the generator wants to minimize it (hence it is a two-player game)
- Architecture guidelines for stable Deep Convolutional GANs
  - Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
  - Use batchnorm in both the generator and the discriminator.
  - Remove fully connected hidden layers for deeper architectures.
  - Use ReLU or LeakyReLU activation in generator for all layers except for the output, which uses tanh/sigmoid.
  - Use LeakyReLU activation in the discriminator for all layers

- Advantages of WGAN:
  - Better training stability.
  - Loss has meaning so it helps in termination criteria
-
- Disadvantages of WGAN
  - Longer to train.

The full algorithm

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α, the learning rate. c, the clipping parameter. m, the batch size.
n_{critic}, the number of iterations of the critic per generator iteration.
Require: : w_0, initial critic parameters. θ_0, initial generator's parameters.

while θ has not converged **do**
  for $t = 0, ..., n_{\text{critic}}$ **do**
    Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
    Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
    $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
    $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
    $w \leftarrow \text{clip}(w, -c, c)$
  end for
  Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
  $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
  $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
end while

Training GAN

1. **procedure** GAN Training
2. **for** number of training iterations **do**
3. **for** k steps **do**
4. Sample minibatch of m noise samples $\{z^{(1)}, .., z^{(m)}\}$ from noise prior $p_g(z)$
5. Sample minibatch of m examples $\{x^{(1)}, .., x^{(m)}\}$ from data generating distribution $p_{data}(x)$
6. Update the discriminator by ascending its stochastic gradient:

7. $$\nabla_\theta \frac{1}{m}\sum_{i=1}^{m}\left[\log D_\theta\left(x^{(i)}\right) + \log\left(1 - D_\theta\left(G_\phi\left(z^{(i)}\right)\right)\right)\right]$$
8. **end for**
9. Sample minibatch of m noise samples $\{z^{(1)}, .., z^{(m)}\}$ from noise prior pg(z)
10. Update the generator by ascending its stochastic gradient

$$\nabla_\phi \frac{1}{m}\sum_{i=1}^{m}\left[\log\left(D_\theta\left(G_\phi\left(z^{(i)}\right)\right)\right)\right]$$

11. **end for**
12. **end procedure**