Machine Learning 410

Lesson 4

Convolutional Neural Networks and Feature Learning

Steve Elston

Reminders

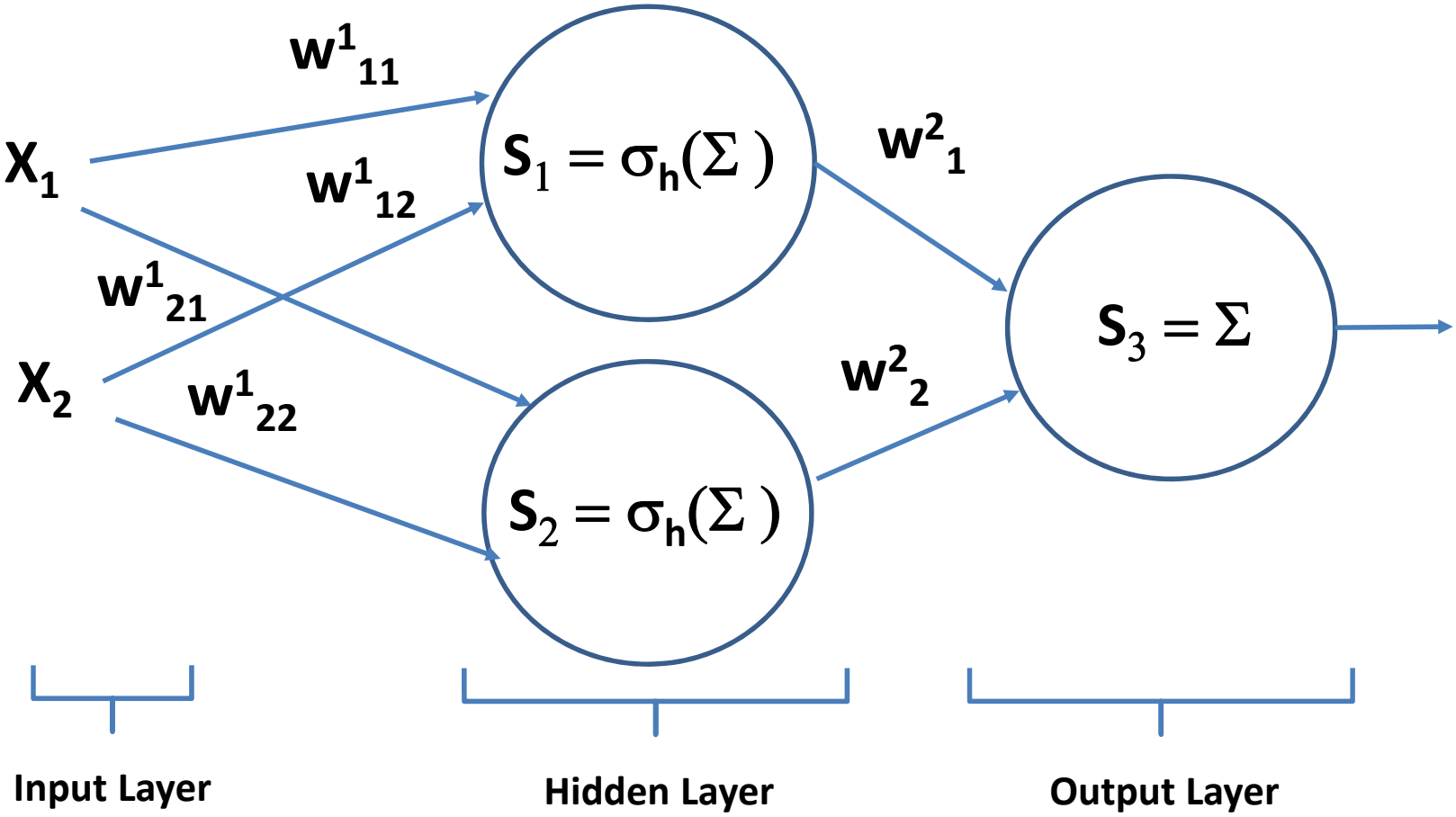
- Discussion – get an easy 5 points!
  - Discussion 3 closes tonight, Oct 17. Don't miss this!
  - Discussion 4 closes Thursday, Oct 14.
  - Discussion 5 closes Thursday, Oct 31.
- Homework – **Updated homework is in Canvas**
  - Homework 2 due Oct 20
  - Homework 3 due Oct 27
  - Homework 4 due Nov 3
- Materials in course Github repository
  - Notebook with reading
  - Slides

Review

- Forward propagation and inference
- Back propagation and gradients
- Loss functions
- Regularization and the bias-variance trade-off
- Optimization with stochastic gradient descent

Forward Propagation and Inference

Forward propagation achieved by **tensor multiplication**



The Backpropagation Algorithm

To **learn model weight tensor** we must **minimize the loss function** using the **gradient:**

$$W_{t+1} = W_t + \alpha \nabla_W J(W_t)$$

Where**:**

W_t = the tensor of weights or model parameters at step t

$J(W)$ = loss function given the weights

$\nabla_W J(W)$ = gradient of J with respect to the weights W

α = step size or learning rate

Example: Computing a Gradient

Goal is to compute the gradient:

The loss function is:

$$J(W) = -\frac{1}{2}\sum_{l=1}^{n}(y_l - S_{3,l})^2$$

$$\frac{\partial J(W)}{\partial W} = \begin{bmatrix} \frac{\partial J(W)}{\partial W_{11}^2} \\ \frac{\partial J(W)}{\partial W_{12}^2} \\ \frac{\partial J(W)}{\partial W_{21}^2} \\ \frac{\partial J(W)}{\partial W_{22}^2} \\ \frac{\partial J(W)}{\partial W_{1}^1} \\ \frac{\partial J(W)}{\partial W_{2}^1} \end{bmatrix}$$

Loss Functions for Training Neural Networks

Given: $\mathbb{D}_{KL}(P \parallel Q) = \mathbb{H}(P) + \mathbb{H}(P, Q)$

The term $\mathbb{H}(P)$ is constant

So, we only need the **cross entropy** term:

$$\mathbb{H}(P, Q) = -\sum_{i=1}^{n} p(x_i)\, ln_b\, q(x_i)$$

The Bias-Variance Trade-Off

- High capacity models fit training data well
  - Exhibit high variance
  - Do not generalize well; exhibit **brittle behavior**
  - $\text{Error}_{\text{training}} << \text{Error}_{\text{test}}$
- Low capacity models have high bias
  - Generalize well
  - Do not fit data well
- Regularization adds bias
  - Strong regularization adds significant bias
  - Weak regularization leads to high variance

The Bias-Variance Trade-Off

- How can we understand the bias-variance trade-off?
- We start with the error:

$$\Delta y = E\left[Y - \hat{f}(X)\right]$$

Where:

$Y =$ the label vector.

$X =$ the feature matrix.

$\hat{f}(x) =$ the trained model.

The Bias-Variance Trade-Off

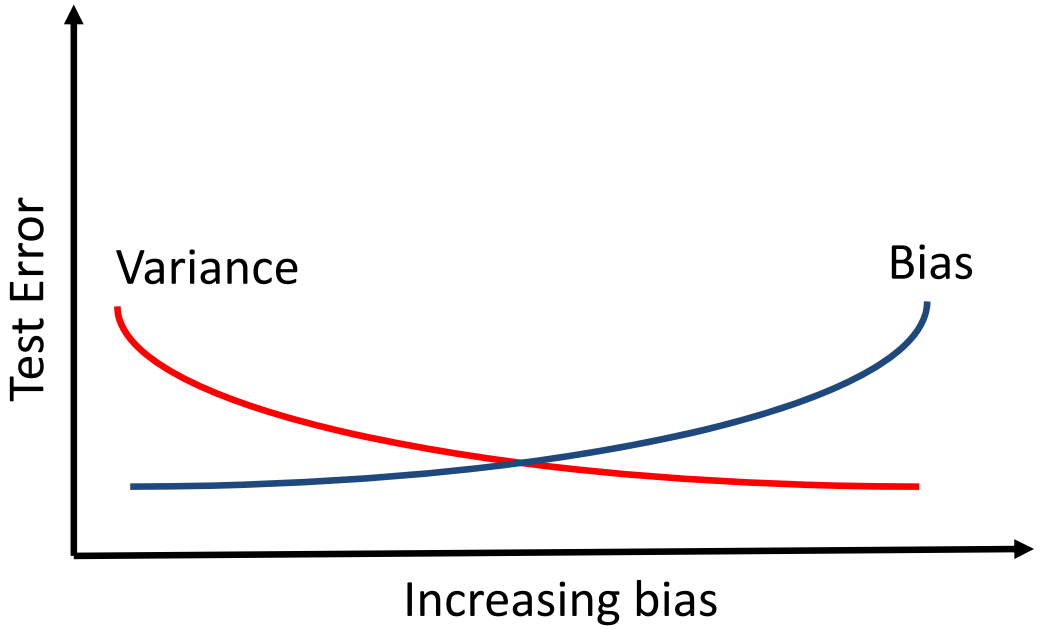
- We can expand the error term

$$\Delta x = \left(E[\hat{f}(X)] - \hat{f}(X)\right)^2 + E\left[(\hat{f}(X) - E[\hat{f}(X)])^2\right] + \sigma^2$$

$$\Delta x = Bias^2 + Variance + Irreducible\ Error$$

- Increasing bias decreases variance
- Notice that even if the bias and variance are 0 there is still irreducible error

The Bias-Variance Trade-Off



l2 Regularization

- One way to limit the size of the model parameters is to constrain the **l2** or **Euclidian norm**:

$$||W||^2 = \left(w_1^2 + w_2^2 + \ldots + w_n^2\right)^{\frac{1}{2}} = \left(\sum_{i=1}^{n} w_i^2\right)^{\frac{1}{2}}$$

- The regularized loss function is then:

$$J(W) = J_{MLE}(W) + \lambda||W||^2$$

- Where λ is the regularization hyperparameter
  - Large λ increases bias but reduces variance
  - Small λ decreases bias and increases variance

Eigenvalues and Regularization

l2 regularization with eigenvalue-eigenvector decomposition

- For the inverse $(A^T A + \alpha^2 \cdot I)^{-1}$ the eigenvalue matrix is:

$$\Lambda^{+}_{Tikhonov} = \begin{bmatrix} \frac{1}{\lambda_1 + \alpha^2} & 0 & 0 & \dots & 0 \\ 0 & \frac{1}{\lambda_2 + \alpha^2} & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & \frac{1}{\lambda_m + \alpha^2} \end{bmatrix}$$

- Even for an eigenvalue, λ, of 0, the biased inverse becomes $1/\alpha^2$
- Thus, the bias term, α^2, creates a 'ridge' of nonzero values on the diagonal ensuring the inverse exists and is stable.

l1 Regularization

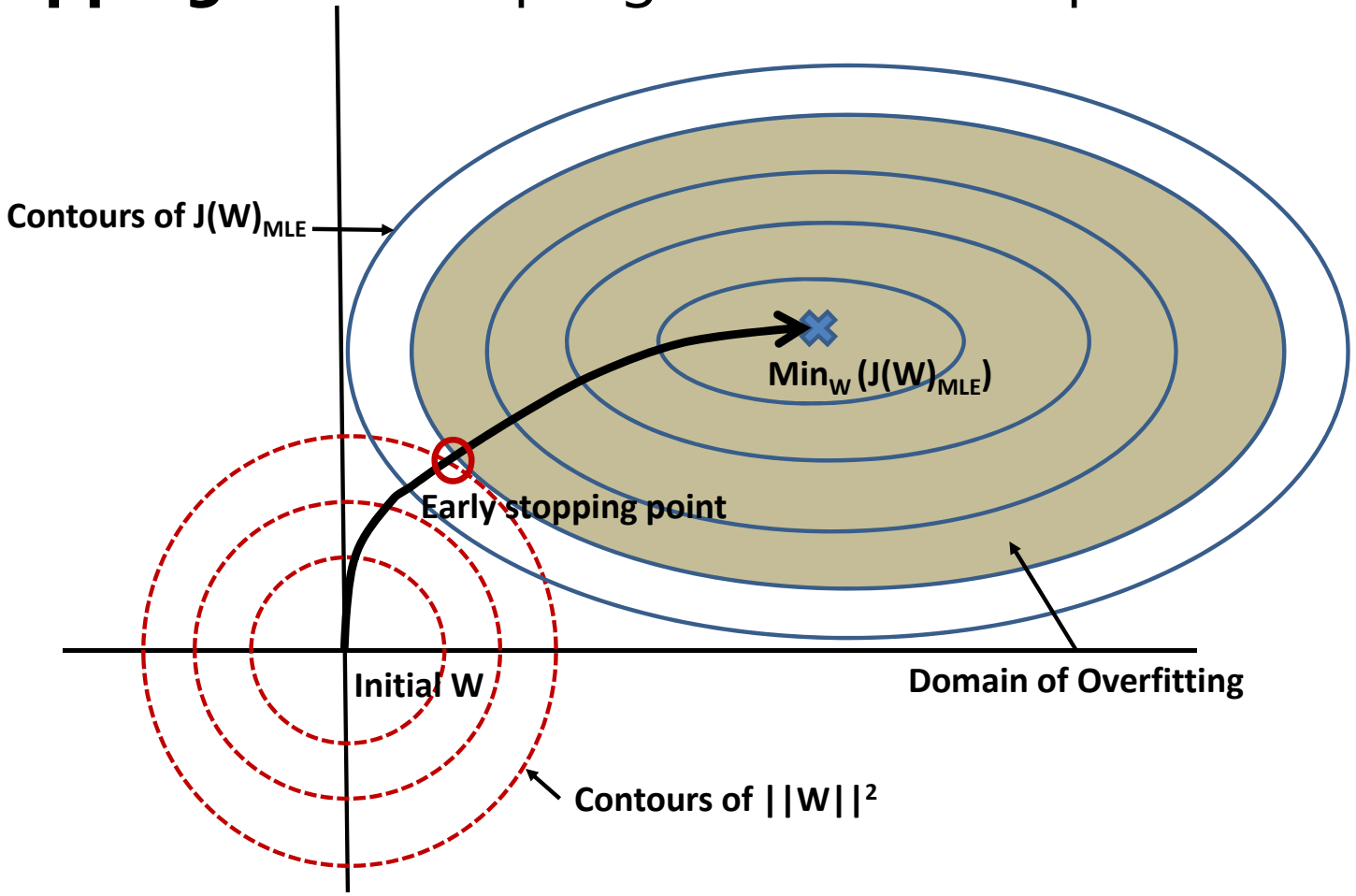
- Given the l1 norm of the weights, the loss function becomes:

$$J(W) = J_{MLE}(W) + \alpha ||W||^1$$

- Where α is the regularization hyperparameter
  - Large α increases bias but reduces variance
  - Small α decreases bias and increases variance
- The l1 constraint drives some weights to exactly 0
  - This behavior leads to the term **lasso regularization**

Early Stopping

Early stopping has a simple geometric interpretation



Dropout regularization

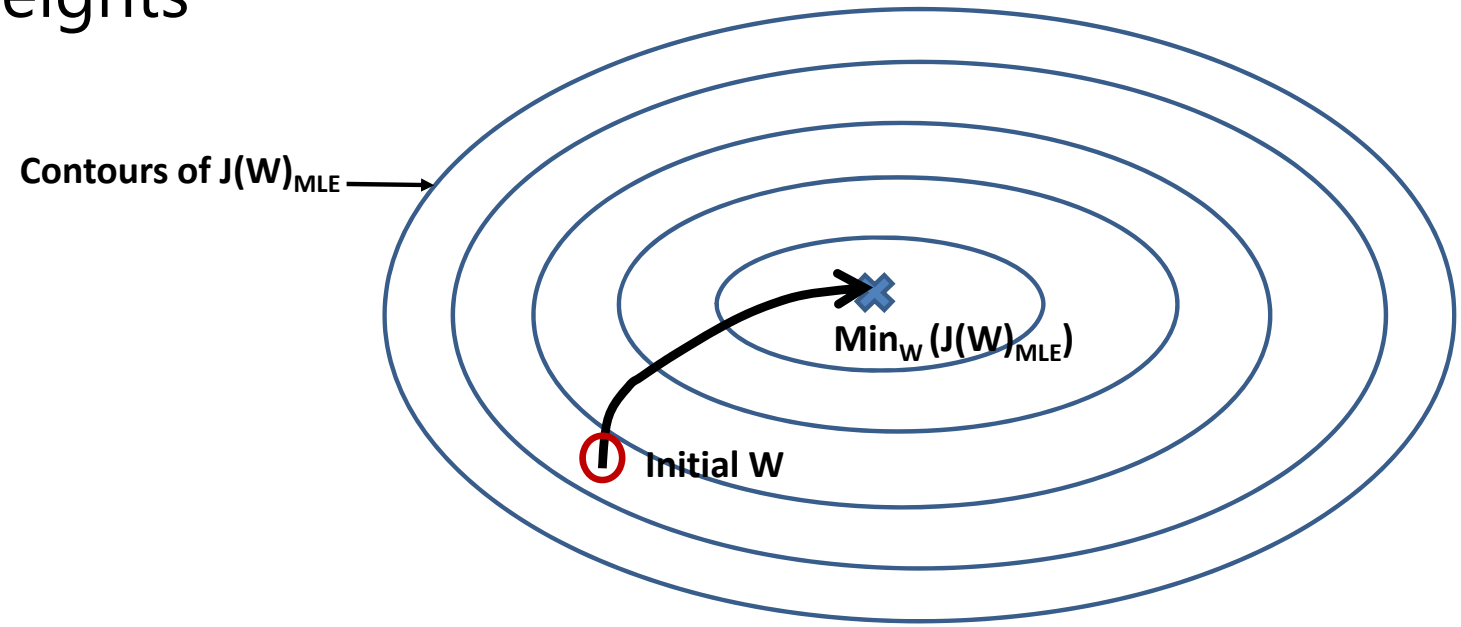
- **Dropout regularization** is a conceptually simple method unique to deep learning
  - At each step of the gradient decent some **fraction, p, of the weights are dropped-out** of each layer
  - The result is a series of models trained for each dropout sample
  - The final model is a **geometric mean** of the individual models
- Weight values are clipped in a small range as a further regularization
- For full details see the readable paper by Srivastava et. al., 2014 http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

Batch Normalization

- In deep neural networks there is a high chance that units in a hidden layer have a **large range of output values**
  - Causes shifts in the covariance of the output values
  - Leads to difficulty computing the gradient
  - Slows convergence for optimizers
- A solution is to normalize the output of the hidden layers in the network as a batch
- This simple idea can be really effective
- For more details see Sergey and Szegedy, 2015: https://arxiv.org/pdf/1502.03167.pdf

Local Convergence of Gradient Descent

- Ideally, the loss function, J(W), is **convex** with respect to the weights



- Convex loss function has **one unique minimum**
- **Convergence** for convex loss function is **guaranteed**

Local Convergence of Gradient Descent

Expand loss function to understand convergence properties of gradient descent:

$$J(W^{(l+1)}) = J(W^{(l)}) + (W^{(l+1)} - W^{(i)})\vec{g} + \frac{1}{2}(W^{(l+1)} - W^{(i)})^T H(W^{(l+1)} - W^{(i)})$$

Where:

$W^{(l)}$ is the tensor of weights at step l

$\vec{g}$ is the gradient vector

H is the **Hessian** matrix.

Local Convergence of Gradient Descent

- How can you understand the Hessian matrix?

$$H\left(f(\vec{x})\right) = \begin{bmatrix} \frac{\partial^2 f(\vec{x})}{\partial x_1^2} & \frac{\partial^2 f(\vec{x})}{\partial x_2 \partial x_1} & \cdots & \frac{\partial^2 f(\vec{x})}{\partial x_n \partial x_1} \\ \frac{\partial^2 f(\vec{x})}{\partial x_1 \partial x_2} & \frac{\partial^2 f(\vec{x})}{\partial x_2^2} & \cdots & \frac{\partial^2 f(\vec{x})}{\partial x_1 \partial x_n} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial^2 f(\vec{x})}{\partial x_1 \partial x_n} & \frac{\partial^2 f(\vec{x})}{\partial x_2 \partial x_n} & \cdots & \frac{\partial^2 f(\vec{x})}{\partial x_n^2} \end{bmatrix}$$

- The Hessian is the matrix of all derivatives of the gradient
- Properties of the Hessian determine convergence rate

Local Convergence of Gradient Descent

- Real-world loss functions are **typically not convex**
- There can be multiple minimums and maximums; a **multi-modal loss function**
- Finding the **globally optimal solution** is hard!
- The minimum reached by an optimizer depends on the **starting value of W**
- In practice, we are happy with a **good local solution**, if not, the globally optimal solution
- Regularization can ‘smooth’ the loss function, at least locally
- First order optimization found to perform as well, or better, than second order

The Nature of Gradients

- Some key properties of the Hessian matrix:
  - The Hessian is symmetric since $\frac{\partial f(\vec{x})}{\partial x_1 \partial x_2} = \frac{\partial f(\vec{x})}{\partial x_2 \partial x_1}$
  - For a convex loss function the Hessian has all **positive eigenvalues**; it is **positive definite**
  - At a maximum point the Hessian has all **negative eigenvalues**; it is **negative definite**
  - The Hessian has **some positive and some negative eigenvalues** at a **saddle point**
  - Saddle points are problematic since direction of descent to the minimum is unclear
  - If Hessian has some **very small eigenvalues**, the gradient is low and **convergence will be slow**

The Nature of Gradients

- For quadratic optimization, the rate of convergence is determined by the **condition number** of the Hessian:

$$\kappa(H) = \frac{|\lambda_{max}(H)|}{|\lambda_{min}(H)|}$$

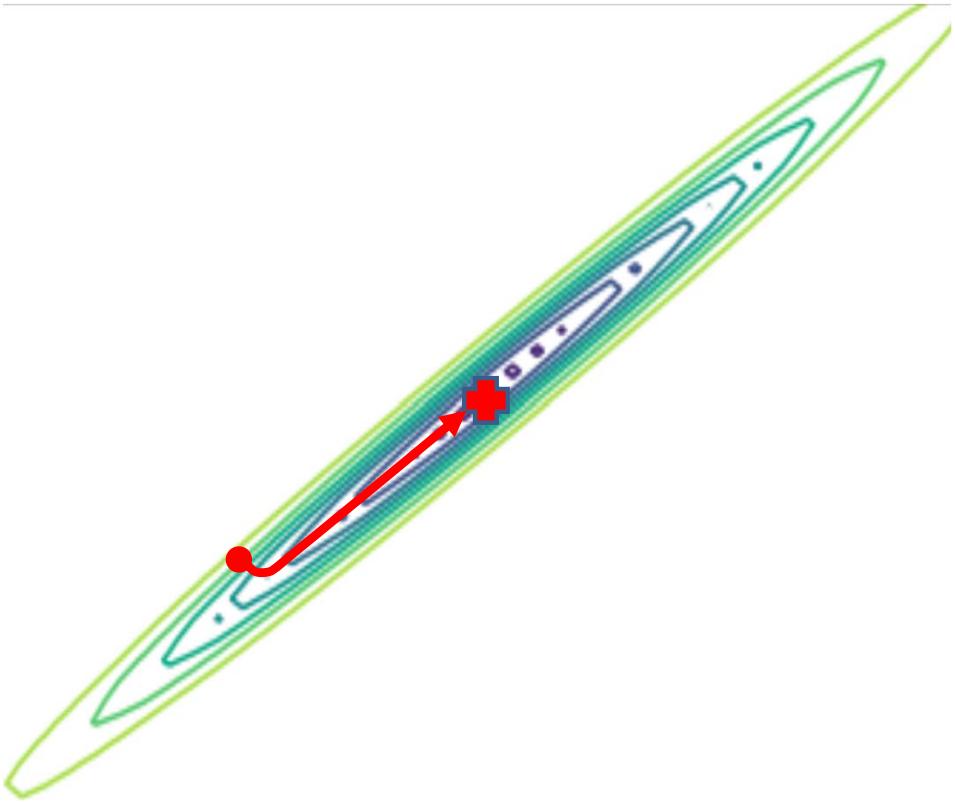
- Where:

$|\lambda_{max}(H)|$ is the absolute value of the largest eigenvalue of H

$|\lambda_{min}(H)|$ is the absolute value of the smallest eigenvalue of H

- If the condition number is close to 1.0, the Hessian is **well conditioned** and convergence will be fast
- If the condition number is large, the Hessian is **ill-conditioned** and convergence will be slow; gradient is flat in some dimensions

Convex vs. Non-Convex Optimization

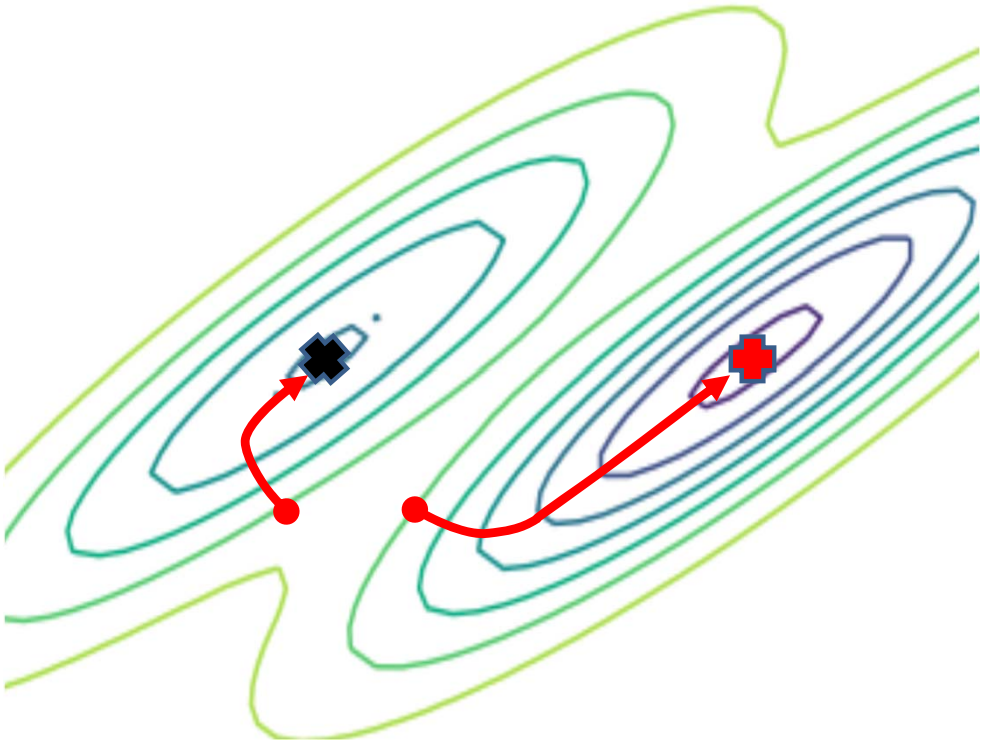


Convex loss
Poorly conditioned

Gradient descent is well-behaved with **convex loss function**

- Only 1 **global minimum**
- From any starting point the gradient leads to global minimum
- Hessian is always **positive definite**

Convex vs. Non-Convex Optimization

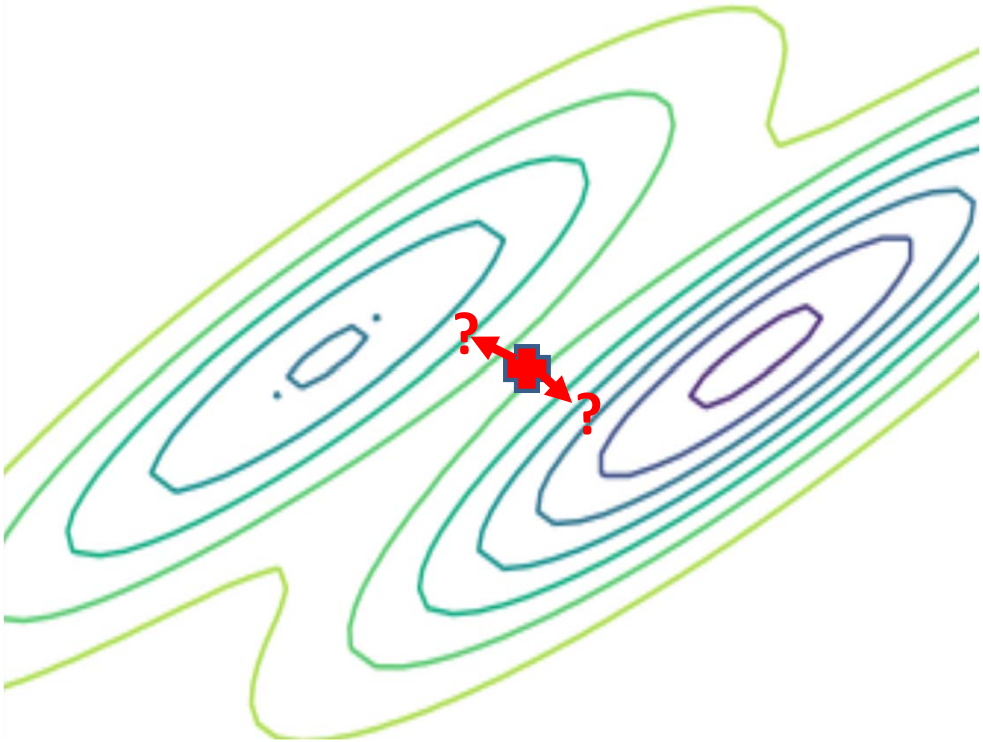


Nonconvex loss

Gradient descent can be problematic with **nonconvex loss function**

- There is a **global minimum**
- Possibly many **local minimums**
- Minimum found with gradient descent depends on starting point
- A good minimum may be good enough
- Hessian positive definite at any minimum, but globally who knows??

Convex vs. Non-Convex Optimization



Nonconvex loss

Gradient descent can be problematic with **nonconvex loss function**

- Can get stuck at **saddle point!**
- Gradient is ambiguous at saddle point
- Hessian is not positive definite

Batch Gradient Descent

- Recall the basic gradient descent equation:

$$W_{t+1} = W_t + \alpha \nabla_W J(W_t)$$

- Where:

W_t = the tensor of weights or model parameters at step t

$\nabla_W J(W)$ = gradient of J with respect to the weights W

α = step size or learning rate

- The gradients of the multi-layer NN are computed using the chain rule

Batch Gradient Descent

- Can we use the gradient descent equation directly?
- Yes, we can
- Iterate the weight tensor relation until a **stopping criteria** or **error tolerance** is reached:

$$||W^{t+1} - W^{t}|| < \text{tolerance}$$

- But;
  - Must compute the gradient for all weights at one time as a **batch** at each step
  - Does not scale if there are a large number of weights

Stochastic Gradient Descent

- We need a more scalable way to apply gradient descent
- **Stochastic gradient descent** is just such a method
- The weight tensor update for stochastic gradient descent follows this relationship:

$$W_{t+1} = W_t + \alpha \, E_{\hat{p}data}\left[\nabla_W J(W_t)\right]$$

Where:

$\hat{p}data$ is the Bernoulli sampled mini-batch

$E_{\hat{p}data}[]$ is the expected value of the gradient given the Bernoulli sample

Stochastic Gradient Descent

- Stochastic gradient descent is known to converge well in practice
- Empirically, using mini-batch samples provide a better exploration of the loss function space
  - Can help solution escape from small local gradient problems
  - Sampling is dependent on mini-batch size

Stochastic Gradient Descent

- Stochastic gradient descent algorithm

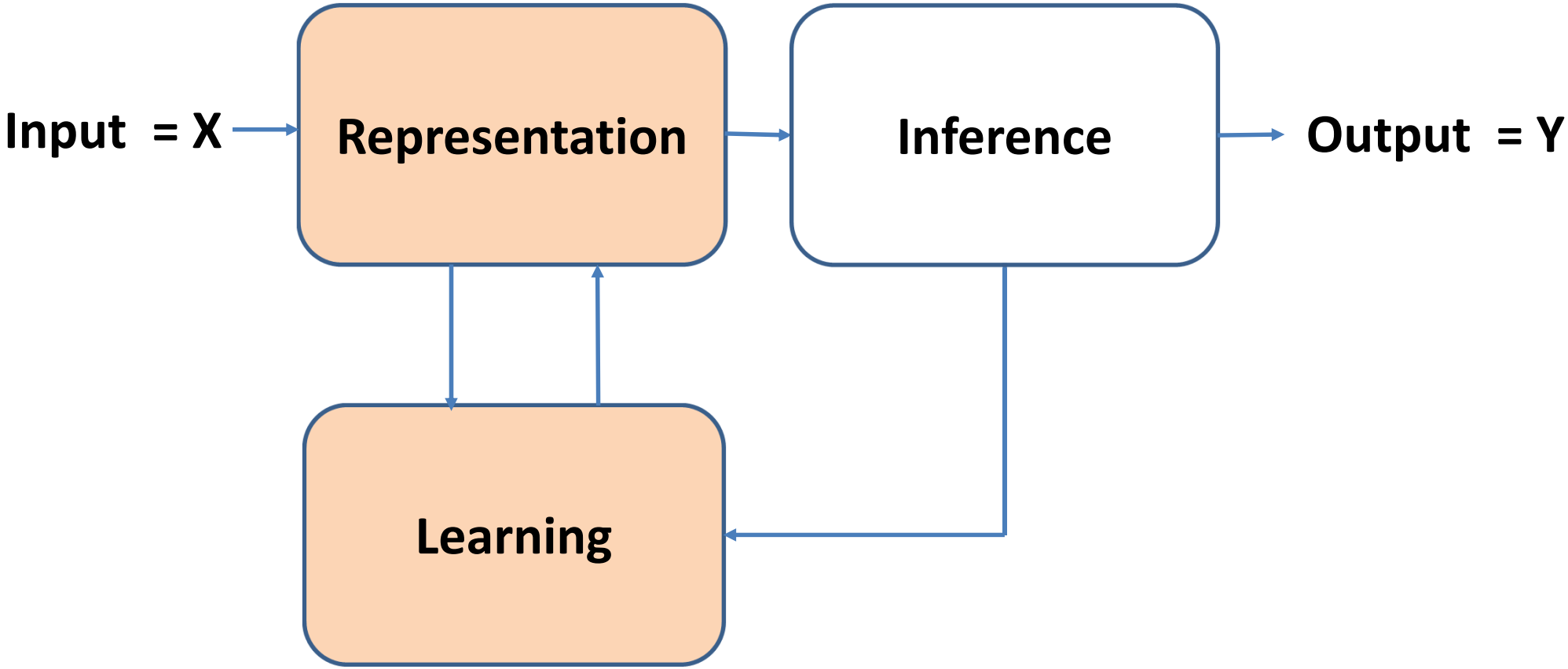
```
Random_sort(cases)
while(grad > stopping_criteria):
    mini-batch = sample_next_n(cases)
    grad = compute_expected_grad(mini_batch)
    weights = update_weights(weights, grad)
```

- Notice that the addition rounds repeat the samples
  - In practice this does not create much bias
  - For large samples this may not happen

Pitfalls in Training Deep Neural Networks

- Fully connected neural networks are **complex function approximators**
  - **Very high dimensional space**
  - **Hard to visualize or understand**
  - Need **regularization to ensure generalization**
  - Gradient descent can find a **good local solution**
- But, training **data may not cover the solution space**
  - **'Holes' in space covered by training** data lead to unexpected results!
  - **More data may not help** if samples have same bias
  - **Biased samples lead to poor generalization**
  - **Regularization can smooth over holes in solution space** and help optimization, at least locally – but **don't expect magic**

Convolutional Neural Networks



Convolutional Neural Networks

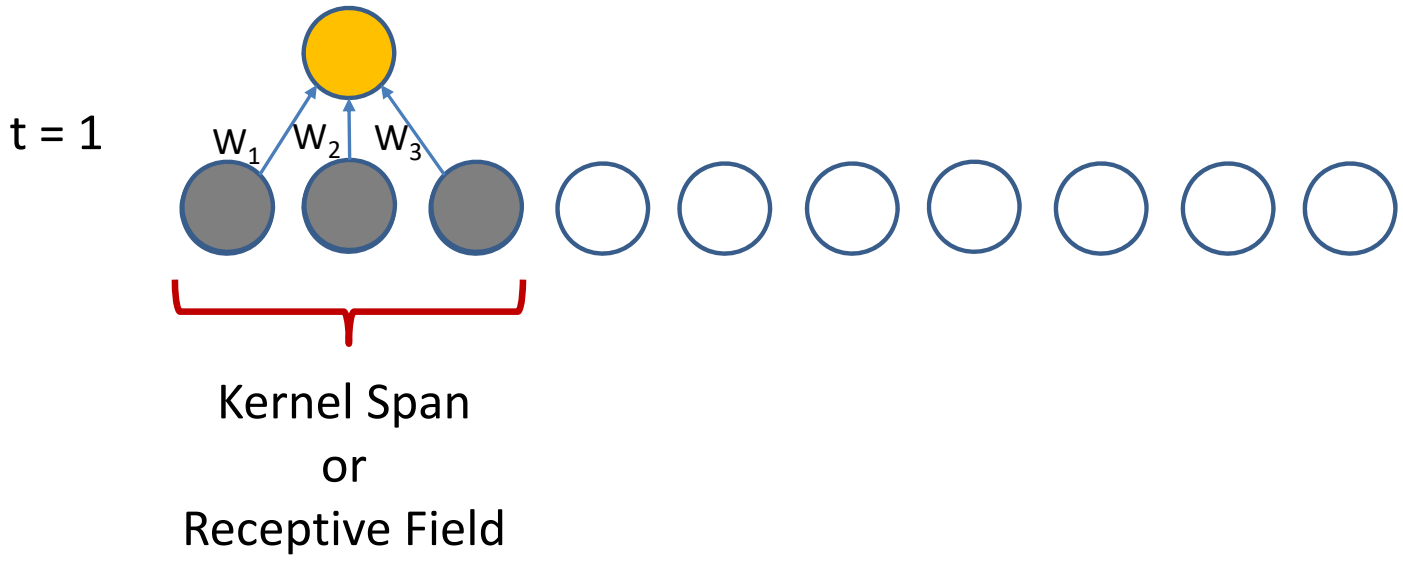
- **CNN**s are used to **learn complex feature maps**
  - **Invariant** to translation and distortion of features
  - **Reduce the dimensionality** of input tensors
  - **Share weights** and are relatively easy to train
- CNNs have a long history
  - LeCun et. al. (1998) first employed CNNs for automatic check handling
  - Era of general use started when Krizhevsky et. al. (2012) won an ImageNet object recognition competition
  - Now commonly used for image, speech and text problems

1-D Convolution

- 1-D CNNs are a simple, but useful example
  - Time series data
  - Text data
  - Speech
- Convolution kernel is moved along the input tensor
  - Kernel has a small span compared to dimension of input tensor
  - At each step a weighted output value is computed

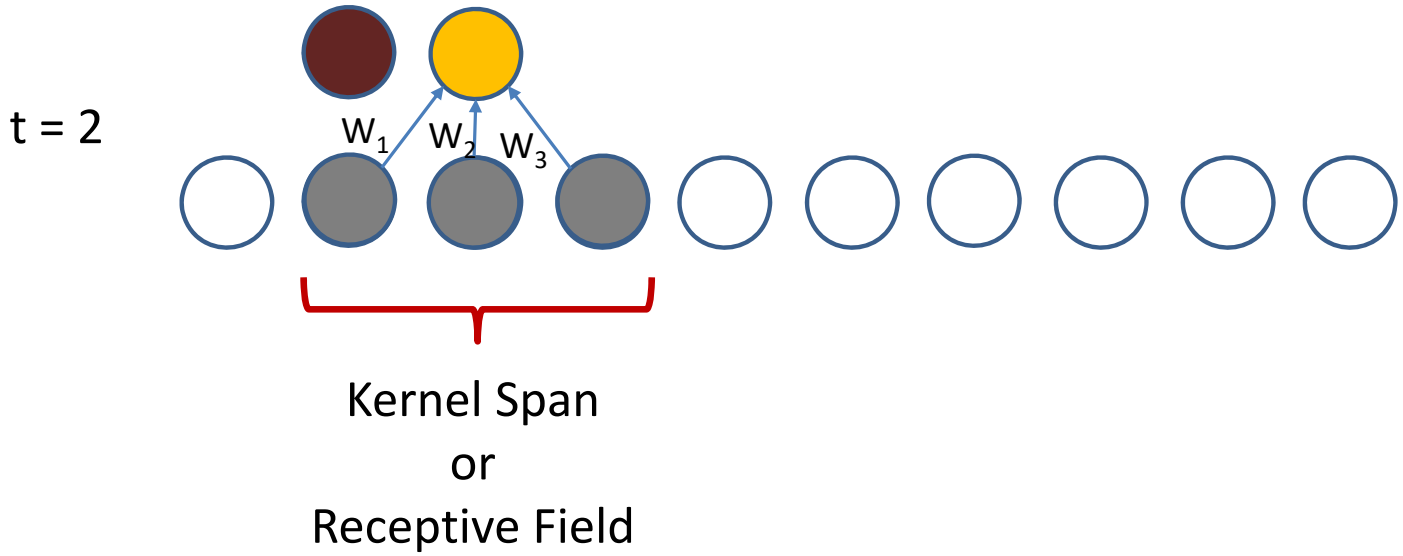
1-D Convolution

1-D CNNs are a simple, but useful example



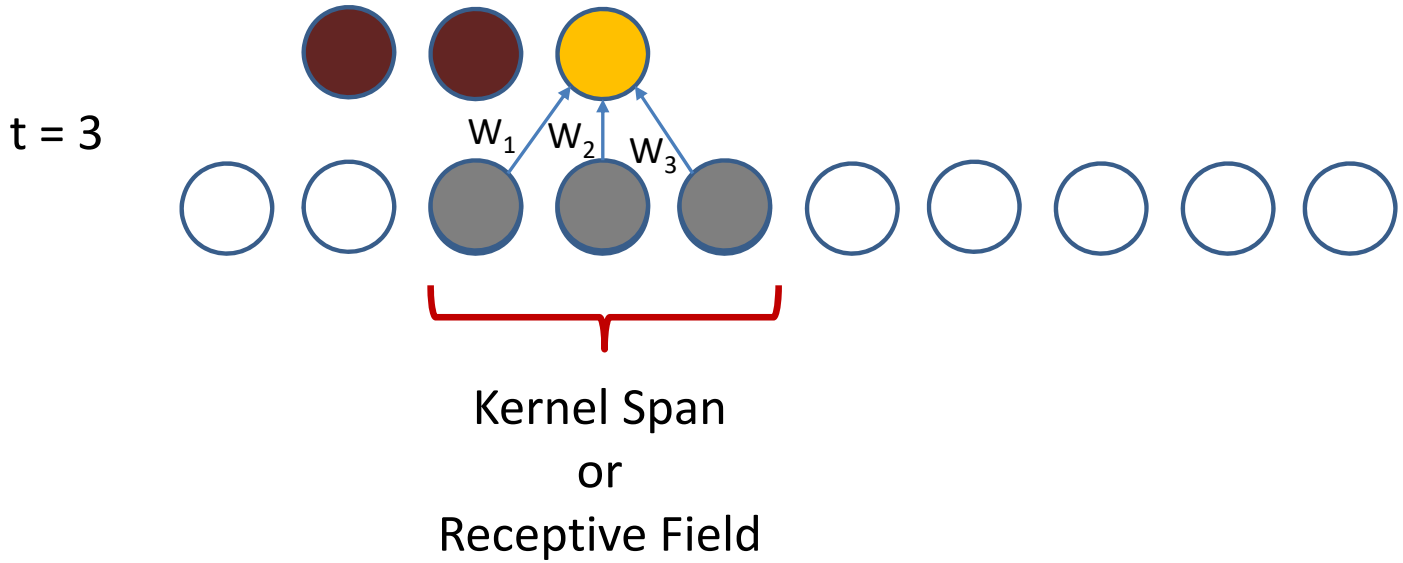
1-D Convolution

1-D CNNs are a simple, but useful example



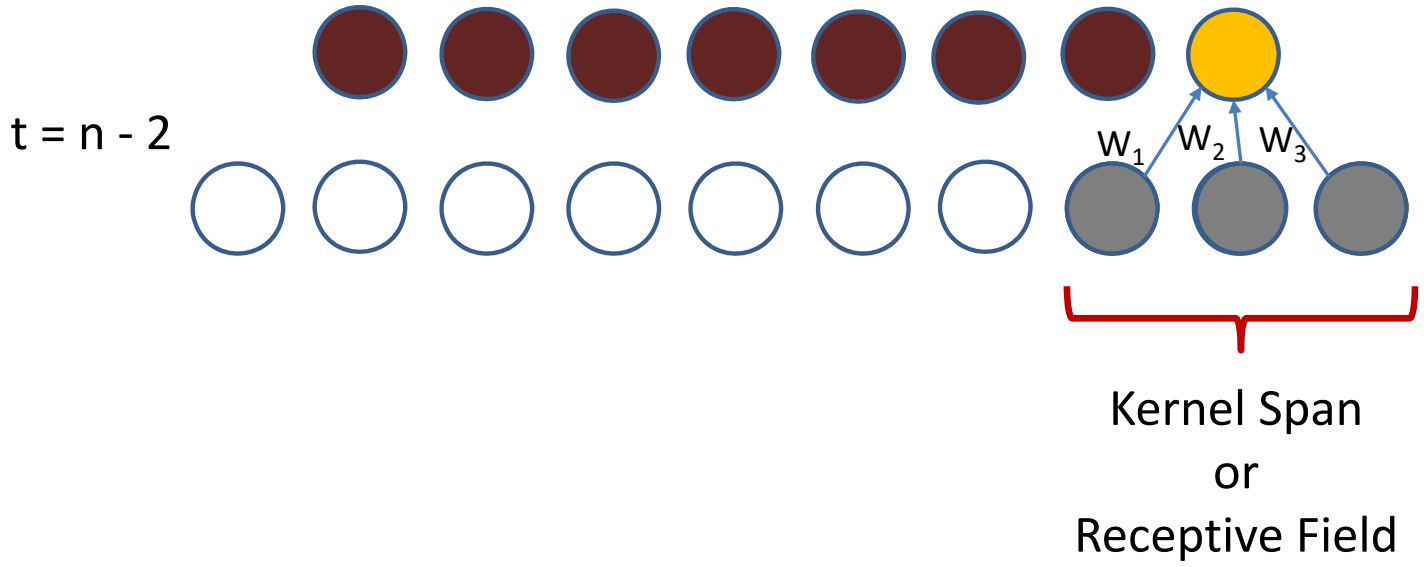
1-D Convolution

1-D CNNs are a simple, but useful example



1-D Convolution

1-D CNNs are a simple, but useful example



1-D Convolution

Mathematically, express 1-d convolution as a weighted sum over a set of discrete kernel values:

$$s(t) = (x * k)(t) = \sum_{\tau=t-a}^{t+a} x(t)k(t-\tau)$$

Where:

$s(t)$ is the output of the convolution operation at time t

x is the series of values,

k is the convolution kernel

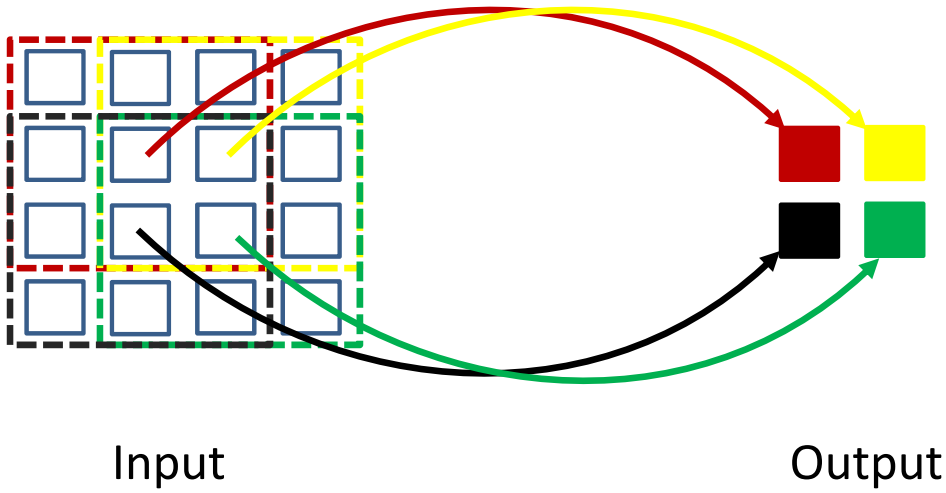
$*$ is the convolution operator

τ is the time difference of the convolution operator.

$a = \frac{1}{2}(kernel\_span + 1)$, for an odd length kernel

2-D Convolution

- 4 x 4 input tensor
- 3 x 3 convolution operator
- 2 x 2 output tensor



2-D Convolution

- Mathematically, we express 2-d convolution as a weighted sum over a discrete rectangular kernel:

$$S(i,j) = (I * K)(i,j) = \sum_{m=i-a}^{i+a} \sum_{n=j-a}^{j+a} I(i,j)K(i-m, j-n)$$

- Where S, I and K are now tensors

2-D Convolution

- The image and kernel tensors are **commutative** in the convolution relationship
- This allows an operation known as **kernel flipping** with the following alternative result:

$$s(i,j) = (I * K)(i,j) = \sum_{m=i-a}^{i+a} \sum_{n=j-a}^{j+a} I(i-m, j-n)K(i,j)$$

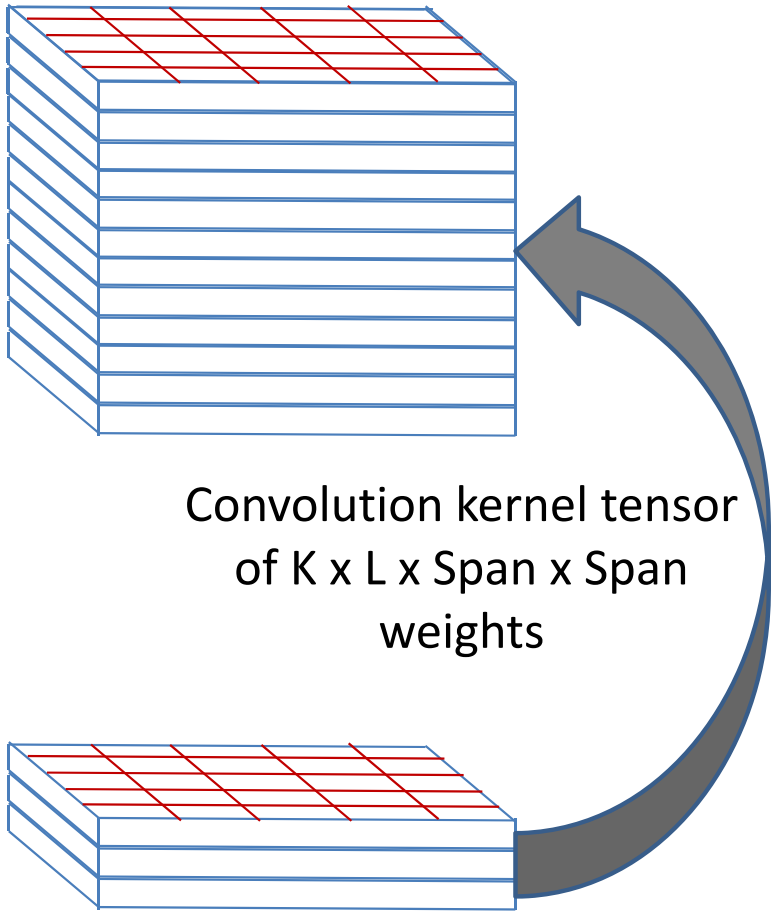
Convolution in Higher Dimensions

- Tensor notation allows easy extension to higher dimensions
  - Input tensor has **multiple input channels**
  - 3-D for color image
  - 4-D for video
- Create **multiple feature maps**
  - Convolution kernel tensor has **multiple output channels**
  - Each output channel is a different feature map
  - Feature in a channel might be vertical lines, horizontal lines, corners, etc

Convolution in Higher Dimensions

Tensor of K output channels

Each channel is a part of **feature map**



Convolution kernel tensor of K x L x Span x Span weights

L input channels
i x j dimensions

Convolution in Higher Dimensions

A multi-dimensional convolution relationship can be written:

$$Z_{i,j,k} = (V * Z)(i,j,k,l) = \sum_{l} \sum_{m=-a}^{a} \sum_{n=-a}^{a} V_{i,j,l} \cdot K_{i-m,j-n,k,l}$$

Where: i, j are the spatial dimenstions

l is the index of the input channel

k is the index of the output channel

$K_{i,j,k,l}$ is the kernel connecting the lth channel of the input to the kth channel of the output for pixel offsets i and j

$V_{i,j,l}$ is the i, j input pixel offsets from channel l of the input,

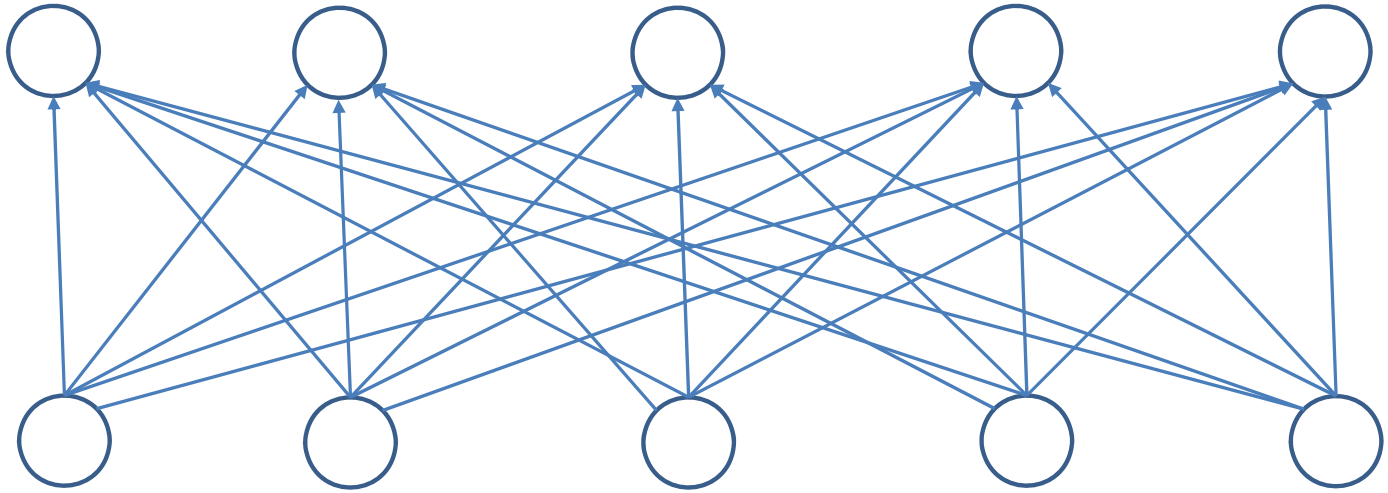
$a = \frac{1}{2}(kernel\_span + 1)$, for an odd length kernel

Parameter Sharing

- The weights of the convolutional kernel must be **learned**
- Each weight of a fully connected network must be learned independently
- CNNs are efficient to train
- CNNs use **parameter sharing**
  - **Statistical strength** from more samples per weight
  - Reduced variance of parameter estimates
- Weights are learned using backpropagation and gradient descent methods
- Also called **tied weights** or **sparce interaction**

Parameter Sharing

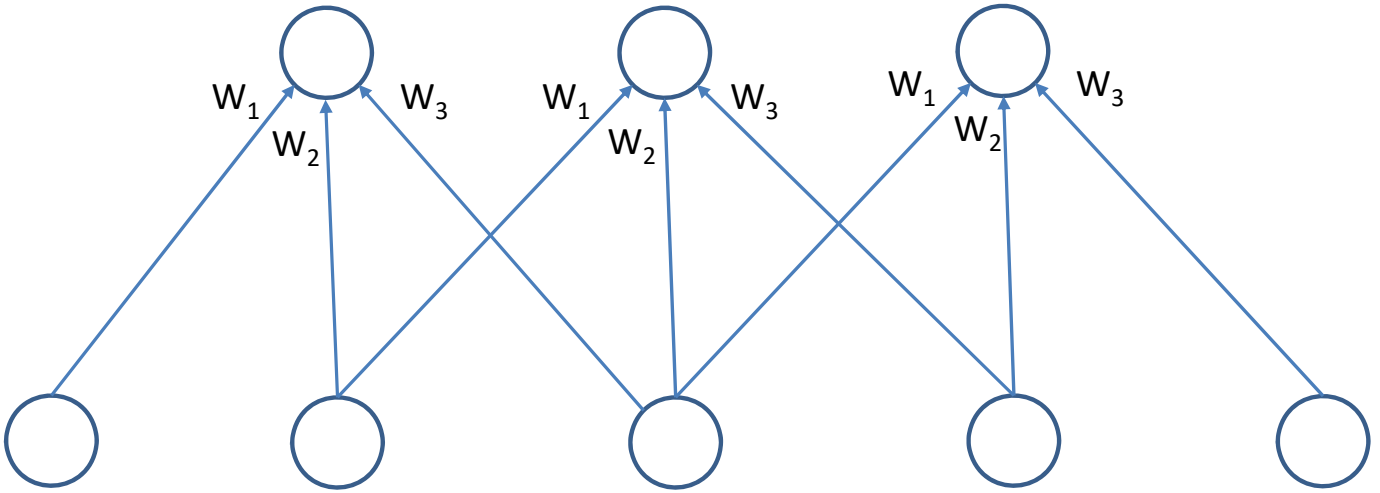
Weights of fully connected network are independent



e.g. requires $5^2 = 25$ weights

Parameter Sharing

Weights are shared in CNN



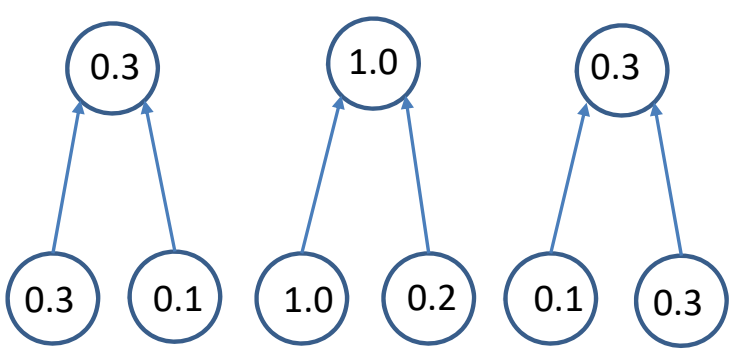
Requires 3 weights

Pooling and Invariance

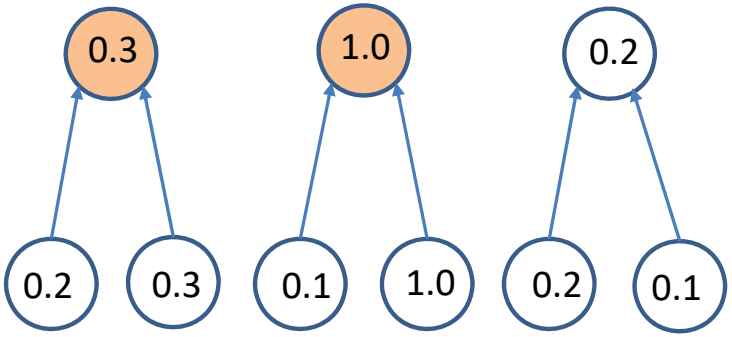
- Convolution provides reduced dimensionality of input tensor
- How can we obtain greater reduction in dimensionality?
- **Pooling** of convolution kernel output values reduces dimensionality
  - e.g. 2x2 operator pools the 4 values into 1
- How to pool?
  - Average?
  - **Max pooling**; simple and highly effective

Pooling and Invariance

Max pooling provides **invariance** to small shifts of the input tensor:



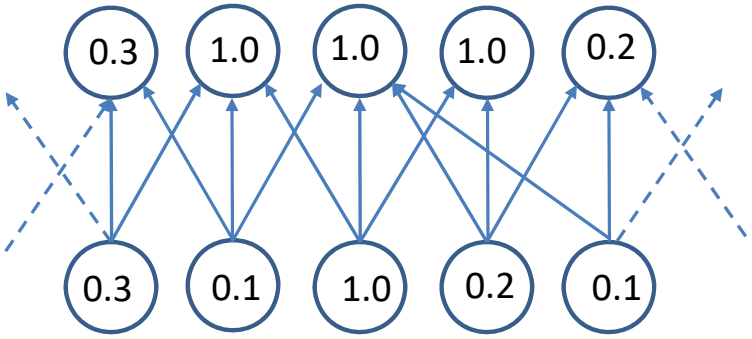
Original



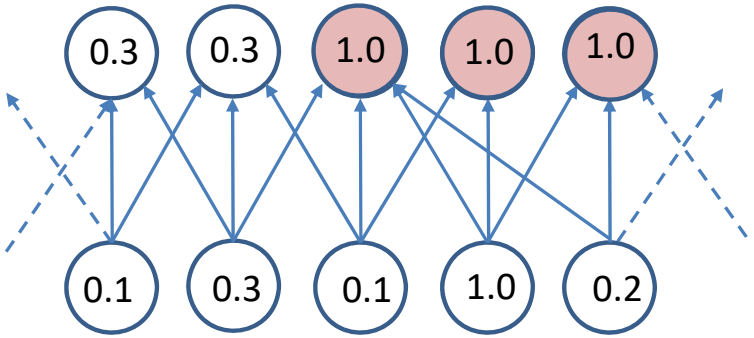
Shift Right by One

Pooling and Invariance

Max pooling provides **invariance** to small shifts of the input tensor:



Original



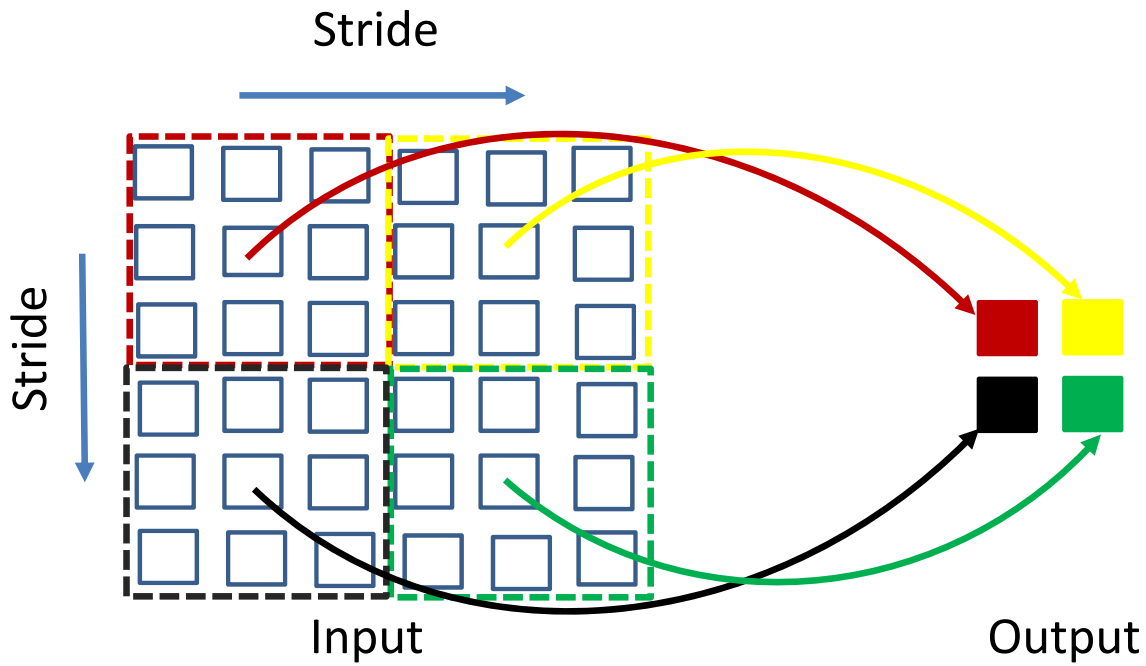
Shift Right by One

Stride and Tiling

- Do we always move the convolution kernel by 1 data value?
- **No!**
  - May not need the full resolution of the input
- We can choose a **stride > 1** for the convolution operator
  - Convolution kernel moved by stride at each step
- Stride > 1 **down-samples** the data
  - Reduces dimensionality
- Stride < 1 up-samples data
  - Useful to create tensor of required dimension

Stride and Tiling

Tiling is a special case when stride = span:



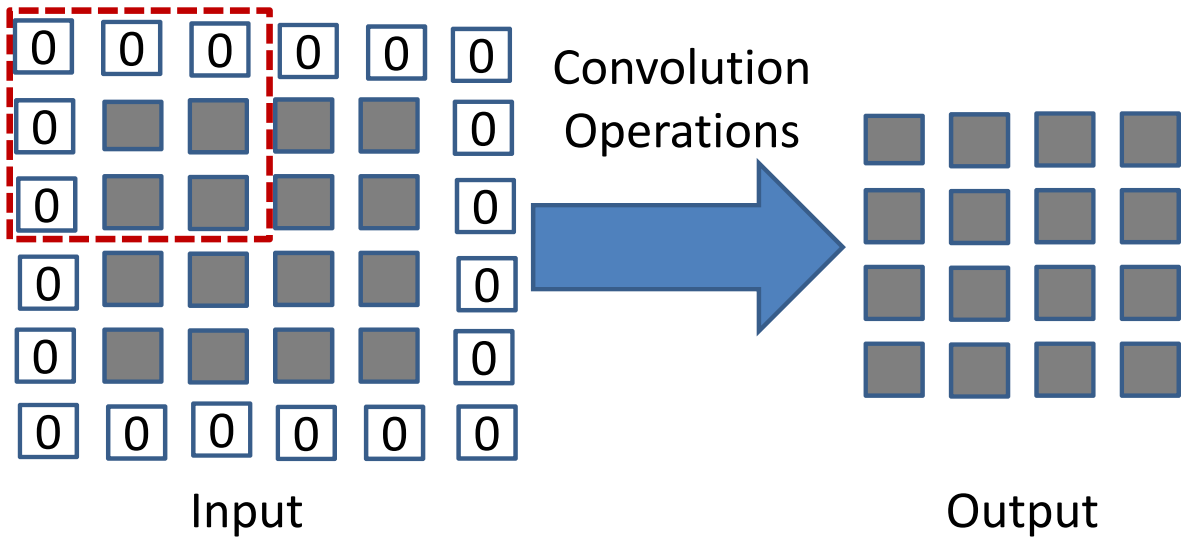
Stride >1 reduces dimensionality

Padding for Convolution

- How can we best deal with **edges of the input** tensor when performing convolution?
- A **valid convolution** confines the kernel to the input tensor dimensions
  - For odd kernel shape, output dimension is (span + 1)/2 less than input dimension
  - After many convolution layers, dimension is reduced further
- We can **zero pad** the input tensor
  - Dimensionality is maintained

Padding for Convolution

- Example: 4x4 input tensor with 3x3 kernel

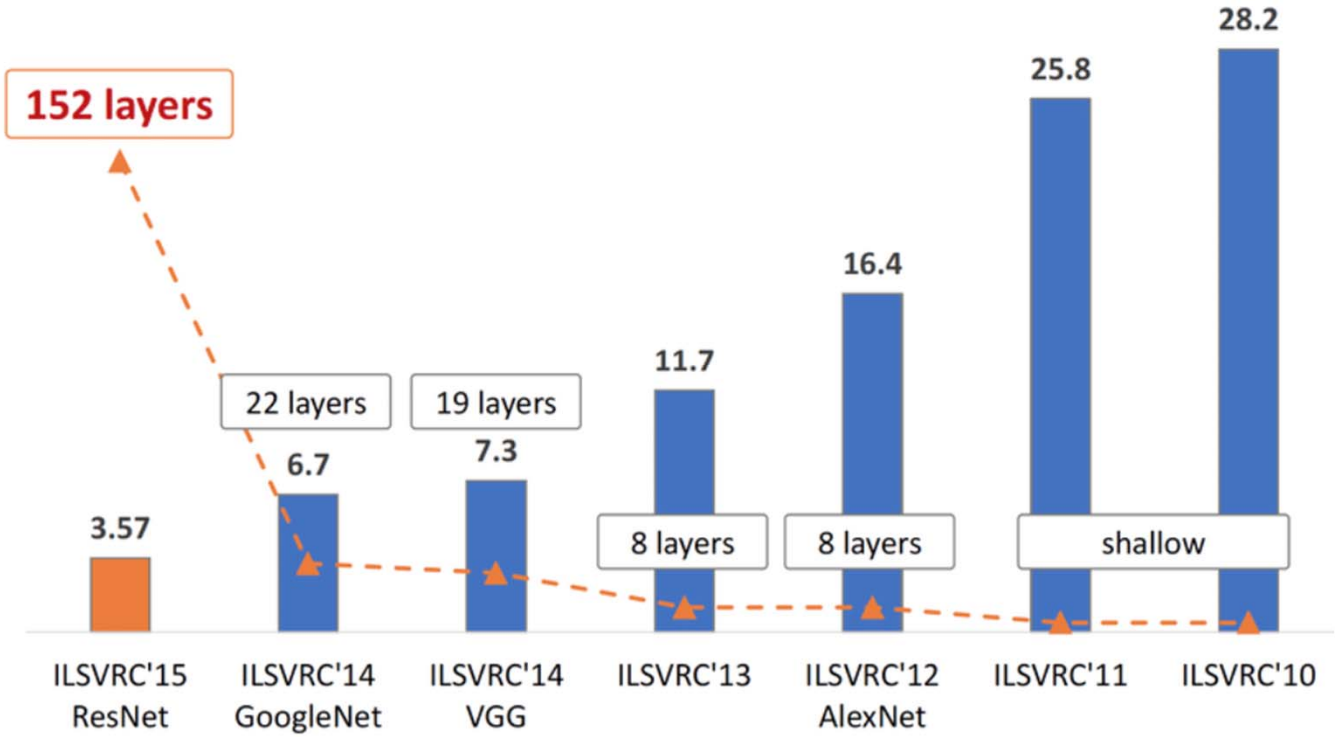


- Dimension of output tensor = dimension of input tensor
- 0 values have little effect with max pooling

Deep and Multi-Scale Architectures

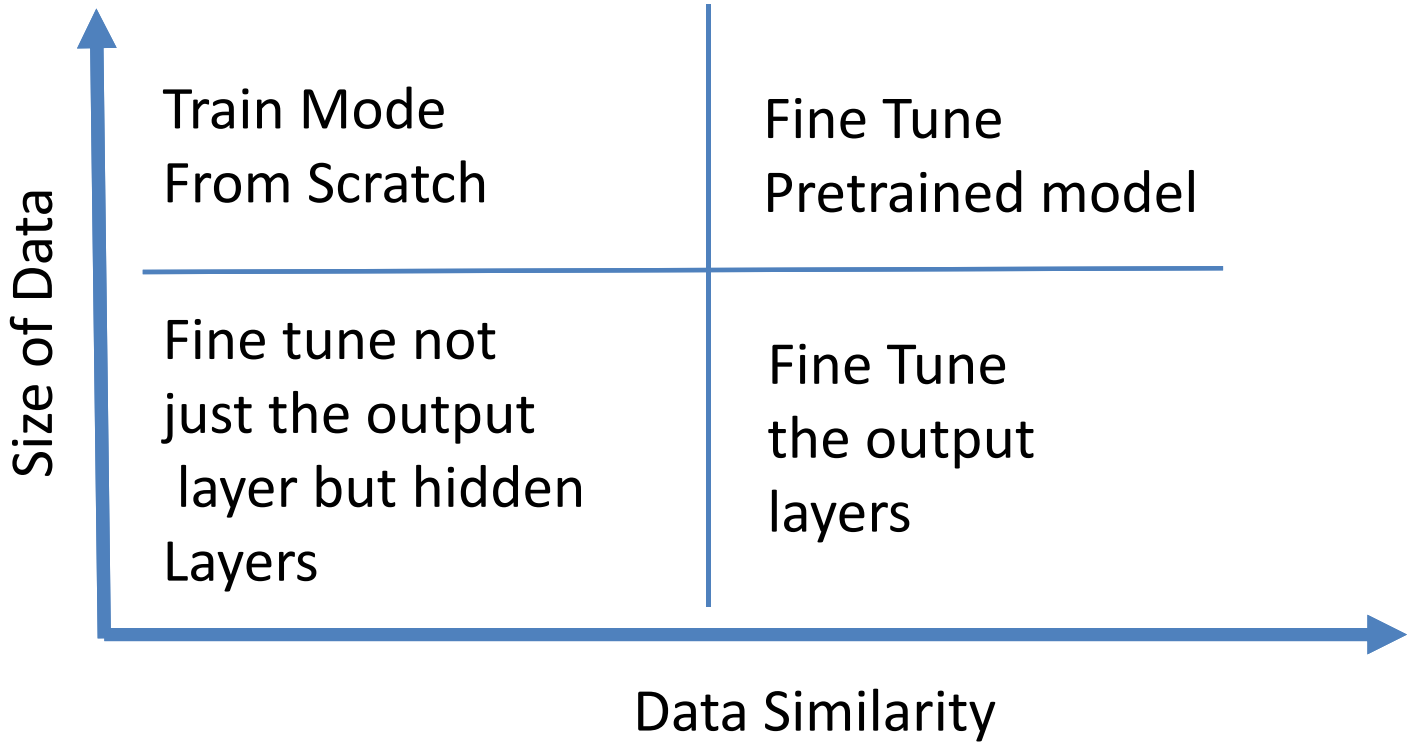
- Deep architectures create large and complex feature maps
- Feature maps have a higher number of channels
- Trained on very large benchmark datasets
- Classification accuracy found to improve with depth

Deep and Multi-Scale Architectures



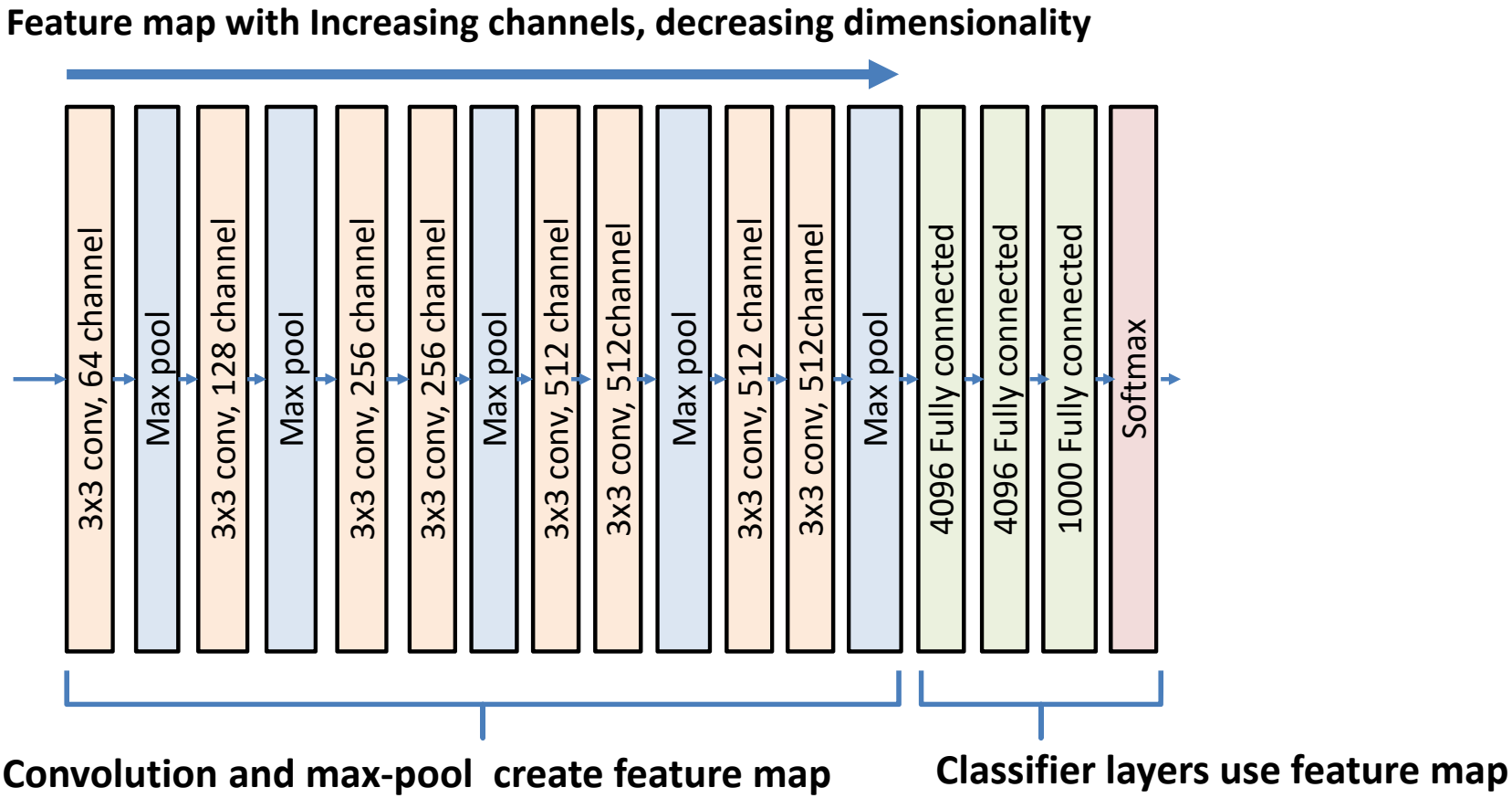
Transfer Learning

- **Transfer learning** exploits learning of a model trained with similar data
- Transfer learning can greatly speed learning for specific solutions



Deep and Multi-Scale Architectures

VGG11 - Simanyan and Zdisserman, 2015



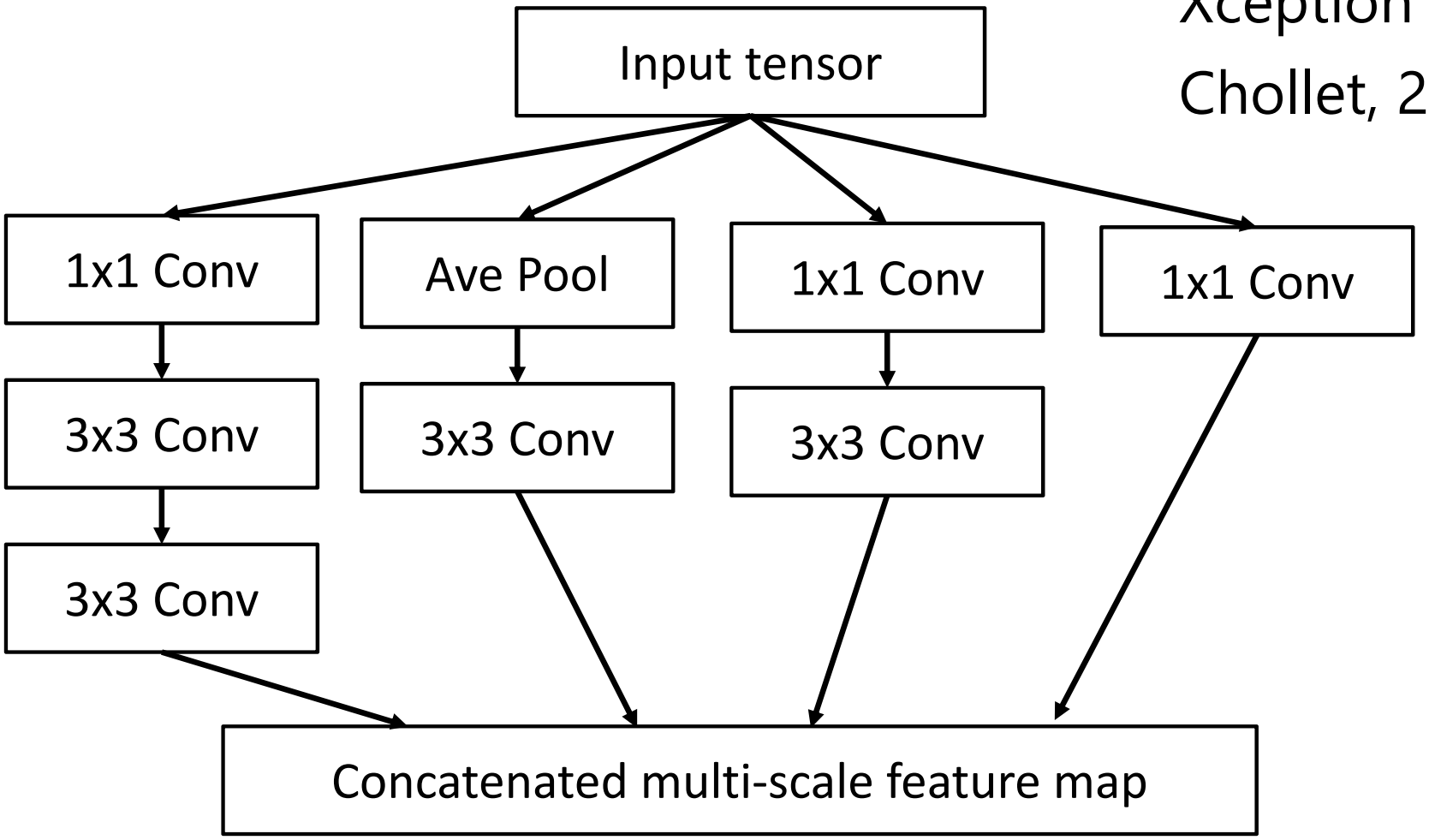
Deep and Multi-Scale Architectures

- Single convolutional layers work at single scale
- But, real-world images contain objects with different scales
- Need architecture that supports multiple scales
  - CNN layers with different scales in parallel
  - Concatenate the results
- Numerous pretrained models, using complex architectures, are now available
  - Trained on very large benchmark datasets
  - Built into deep learning frameworks; Keras, PyTorch, etc.

Deep and Multi-Scale Architectures

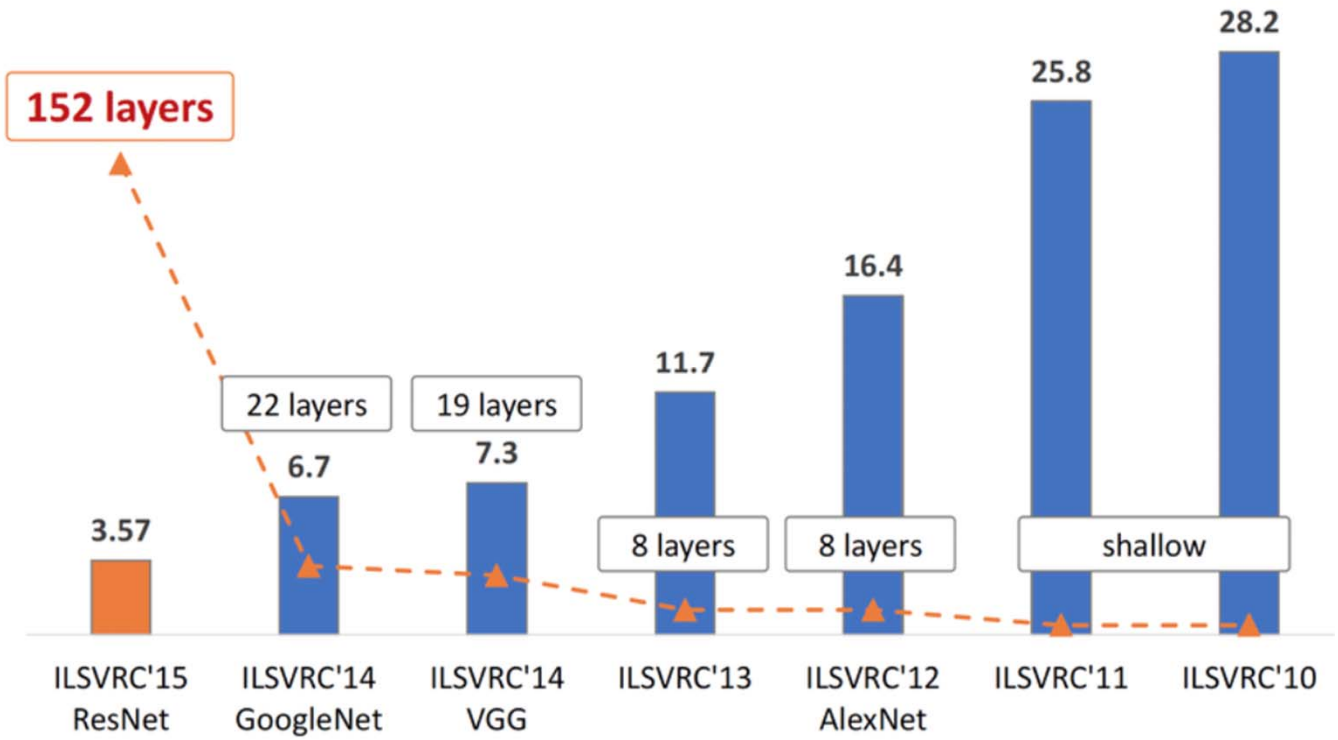
Xception

Chollet, 2017



Deep and Multi-Scale Architectures

- Very deep CNNs can provide significant improvements in accuracy
- But, **learning is hard** in very deep networks



Limitations on Learning with Stacked Layers

What happens when we use backpropagation to train deep NNs with larger number of layers?

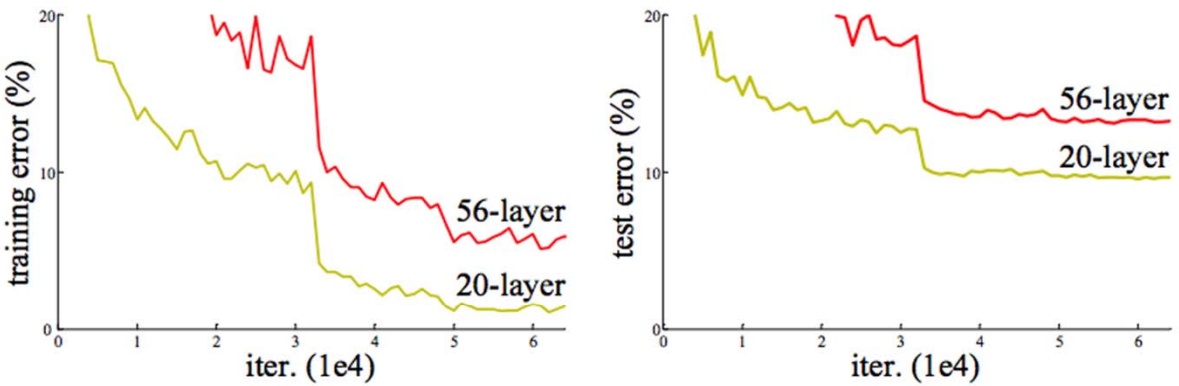


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

https://arxiv.org/pdf/1512.03385.pdf

- How can the behavior in the charts be explained?
  - Vanishing gradients?
  - Exploding gradients?
  - Overfitting?
- None of the above!
  - Vanishing gradients – batch normalization
  - Exploding gradients – gradient clipping
  - Overfitting – dropout regularization

Limitations on Learning with Stacked Layers

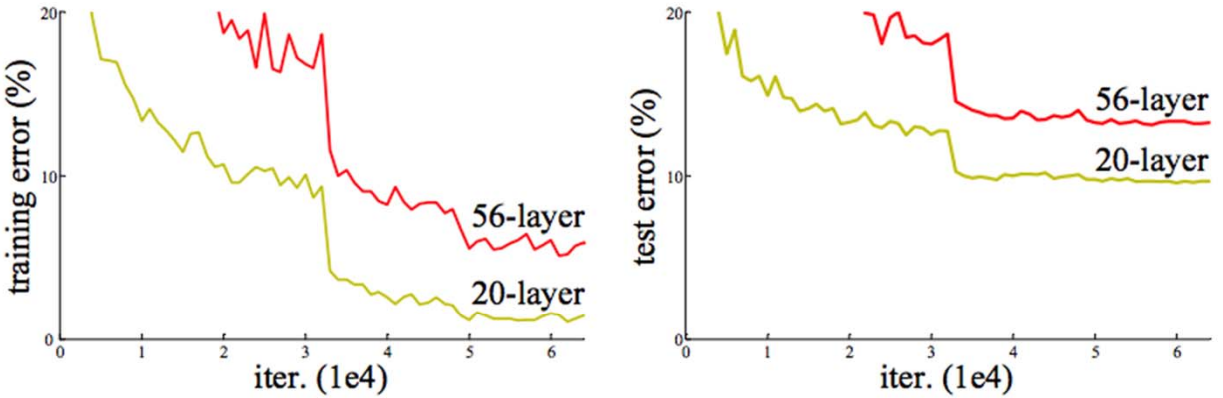


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

- What can explain this result?
- There must be a limitation with the representation!
- The learning with backpropagation on stacked layers no longer learns useful features past a certain depth

Limitations on Learning with Stacked Layers

Idea: Try to learn a different function

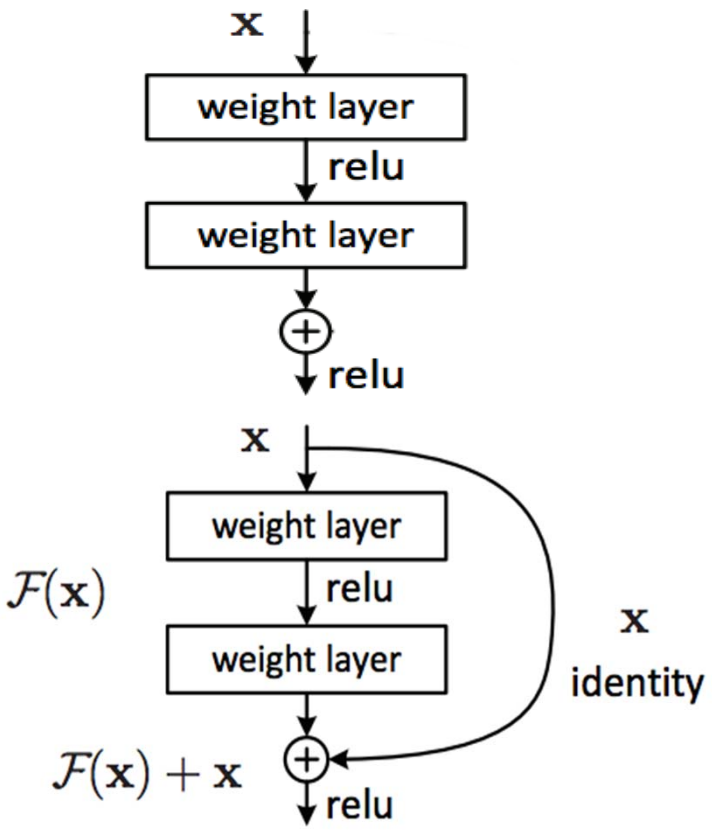


Figure 2. Residual learning: a building block.

- Ideally a subset of stacked layers learns a nonlinear function, $\mathcal{H}(x)$
- But, can also learn a **residual function**

$$\mathcal{F}(x) = \mathcal{H}(x) - x$$

- Rearrange terms to derive a different **residual mapping** function

$$y = \mathcal{F}(x) + x$$

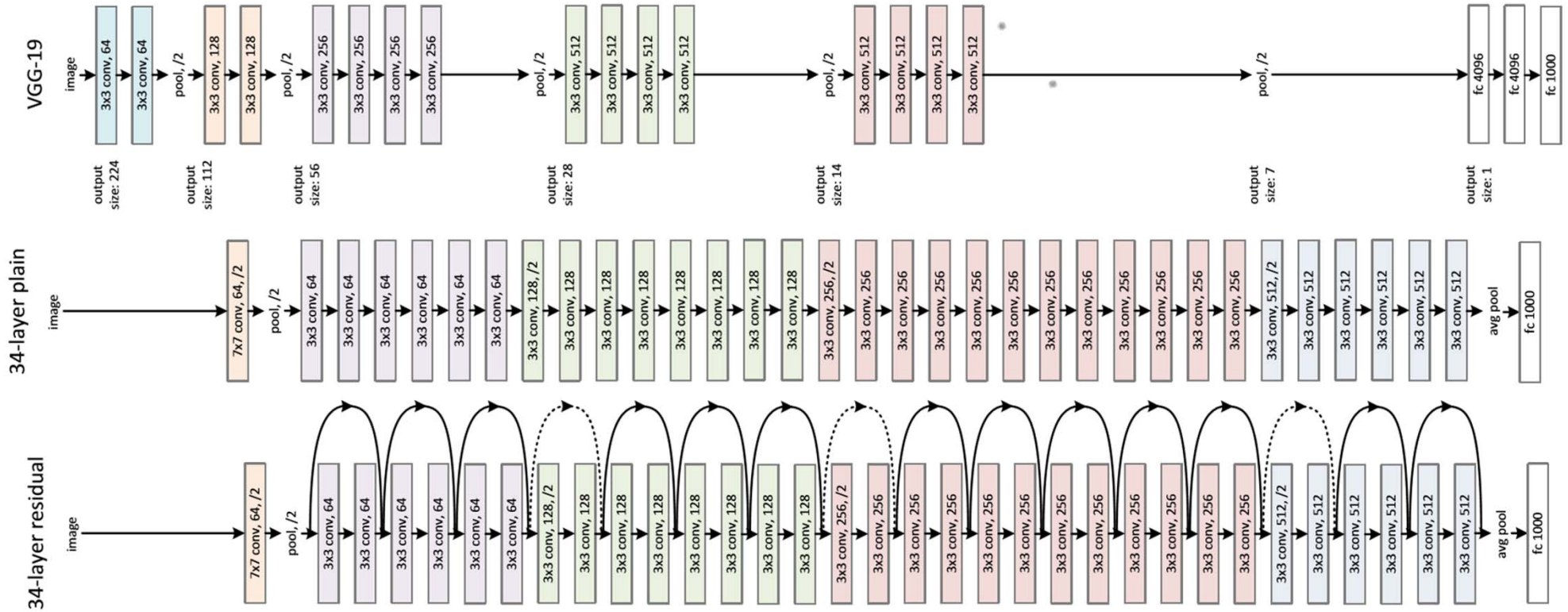
- Or with explicitly showing weights,

$$y = \mathcal{F}(x, \{Wi\}) + x$$

- Layers now have an **identity shortcut**

ResNet Representation with Residuals

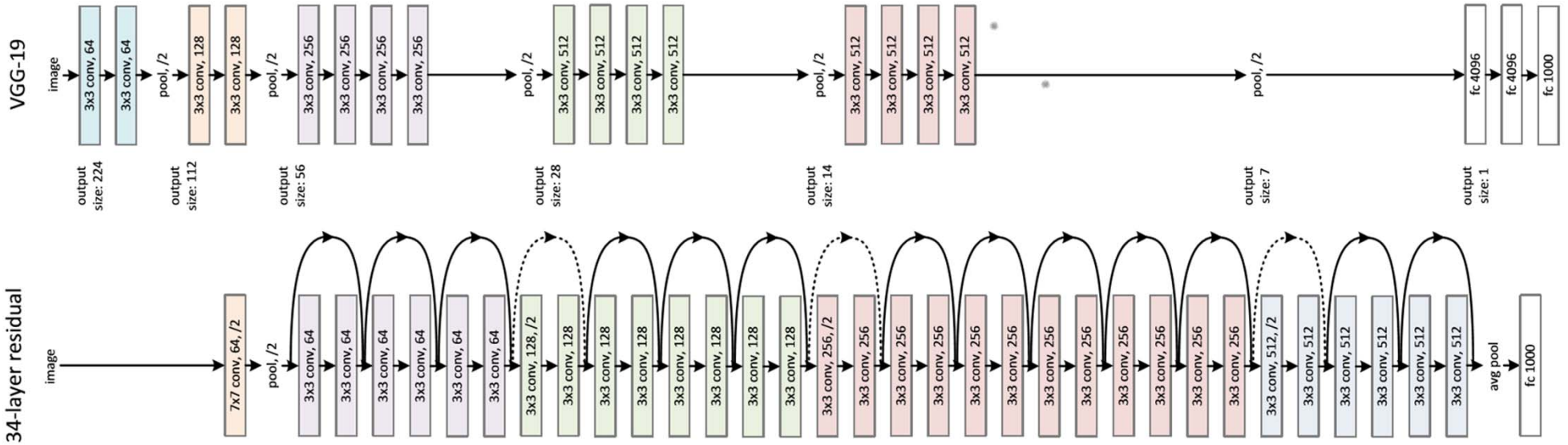
- Start with VGG19 architecture
- 34 stacked layer network
- 34 layer ResNet with **identity shortcuts**



ResNet Representation with Residuals

How does computational complexity compare?

- Notice ResNet has **only 2 max pooling filters**!
- Forward propagation with VGG19 requires 19.6 B Floating Point Operations (GFLOPs)
- Forward propagation with ResNet-34 requires 3.6 GFLOPs!



ResNet Representation with Residuals

How well does ResNet work?

- Compare 18 and 34 layers stacked network performance
- Next, compare 18 and 34 layer ResNets

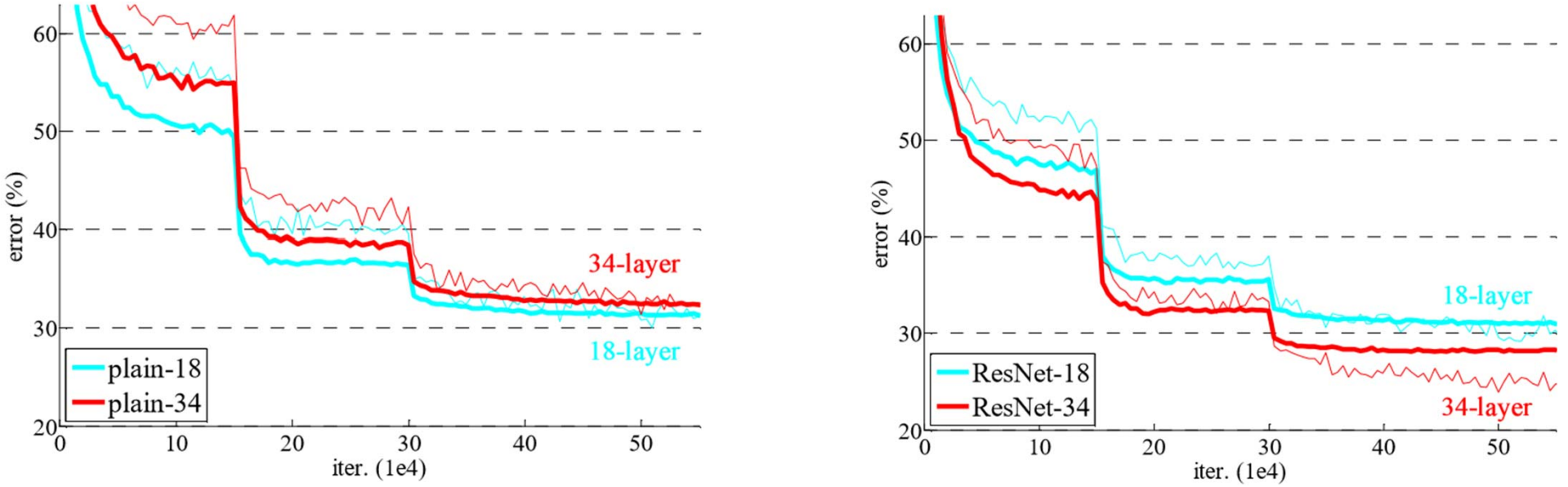


Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

ResNet Representation with Residuals

What are the limits of ResNet Learning?

- Compare stacked network performance drops past about 20 layers
- ResNets allow greater depth and accuracy, but with limits!

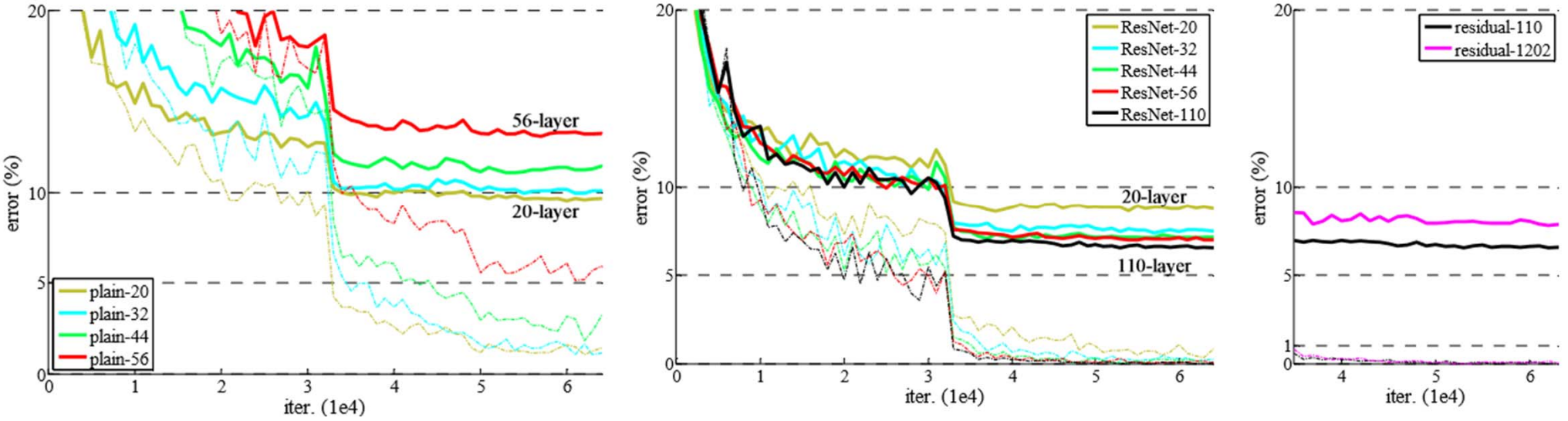
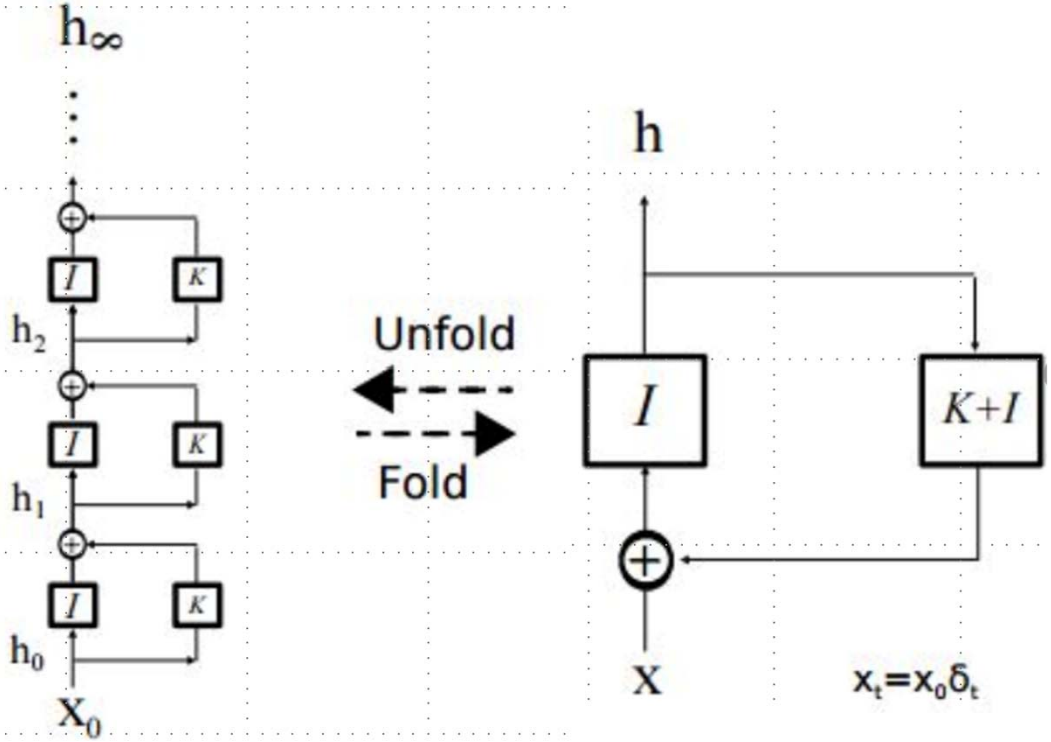


Figure 6. Training on **CIFAR-10**. Dashed lines denote training error, and bold lines denote testing error. **Left**: plain networks. The error of plain-110 is higher than 60% and not displayed. **Middle**: ResNets. **Right**: ResNets with 110 and 1202 layers.

Why do ResNets work?

Why is learning easier with ResNets?



- Start with ResNet feed-forward layers
- Now, **unroll the recurrence** for several learning steps
- Learning in the unrolled model resembles **ensemble learning**

Qianli Liao, Tomaso Poggio,Bridging the Gaps Between Residual Learning, Recurrent Neural Networks and Visual Cortex,

Scaling with ResNets

How do we deal with shortcut connections with different scales of layers?

1. Zero pad
2. Use **linear projection connection** to up or down sample ta

$$y = \mathcal{F}(x, \{Wi\}) + W_s x$$

- Zero pad is nearly as good projection connection, but many fewer weights to learn
- Zero padding used in ResNet

ResNet: Residual Learning Building Block

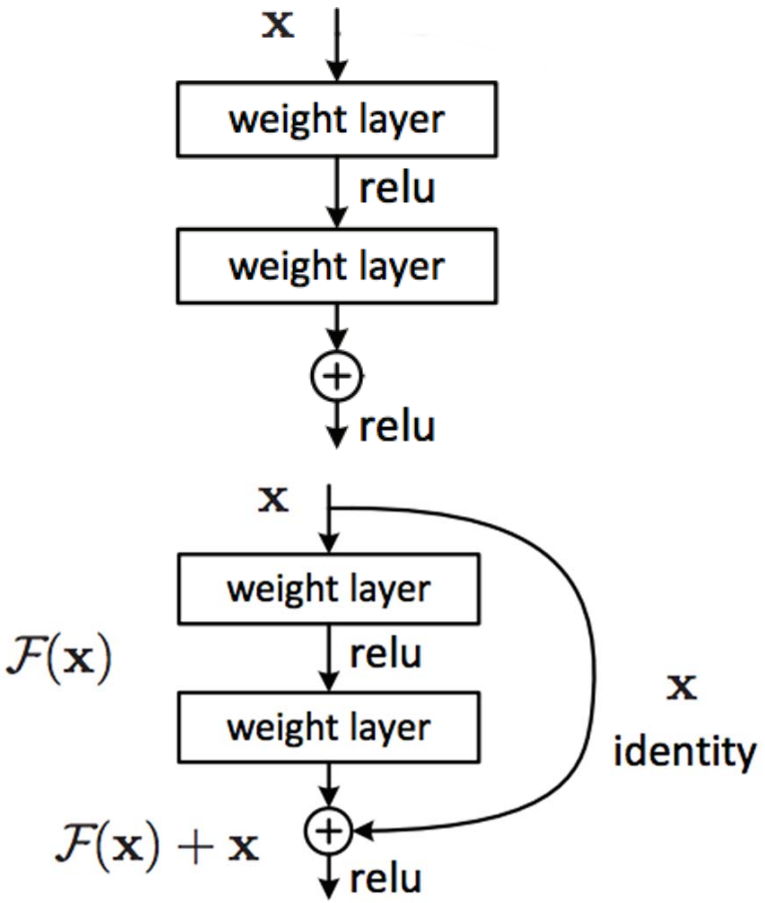
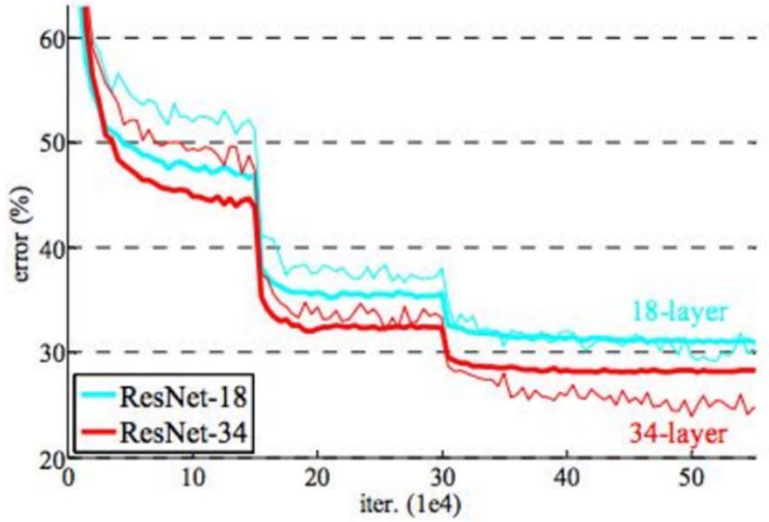


Figure 2. Residual learning: a building block.



With residuals, the 34-layer network outperforms the 18 layer.

0	0	0
0	1	0
0	0	0

With 'SAME' padding, this will output the same feature map it receives as input

LeNet5: Model

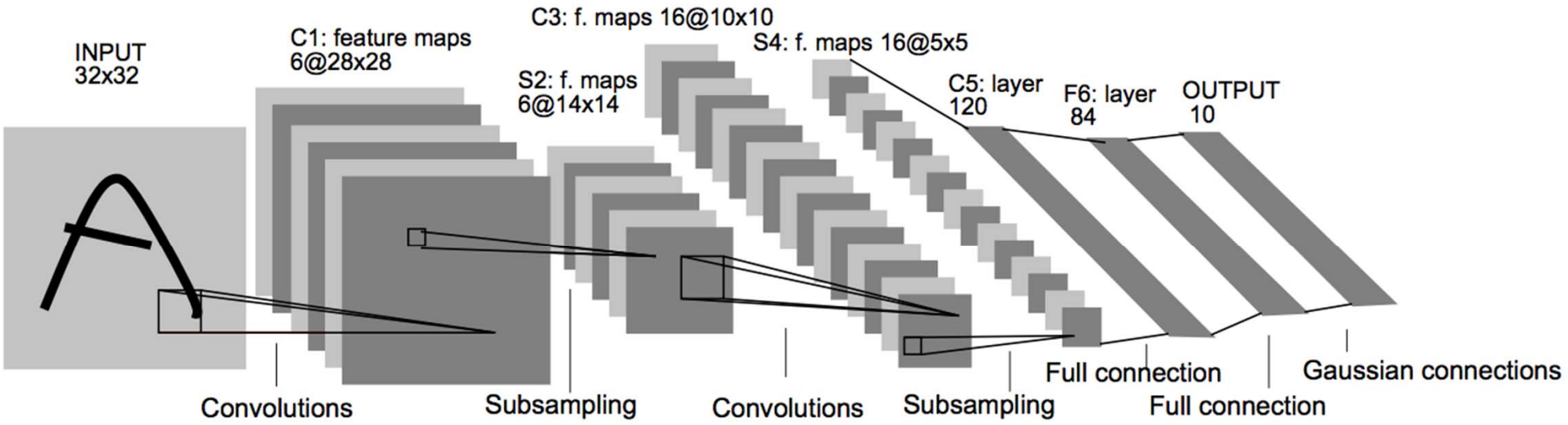


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

```
# import the necessary packages
from keras.models import Sequential
model = Sequential()
```

LeNet5: Input

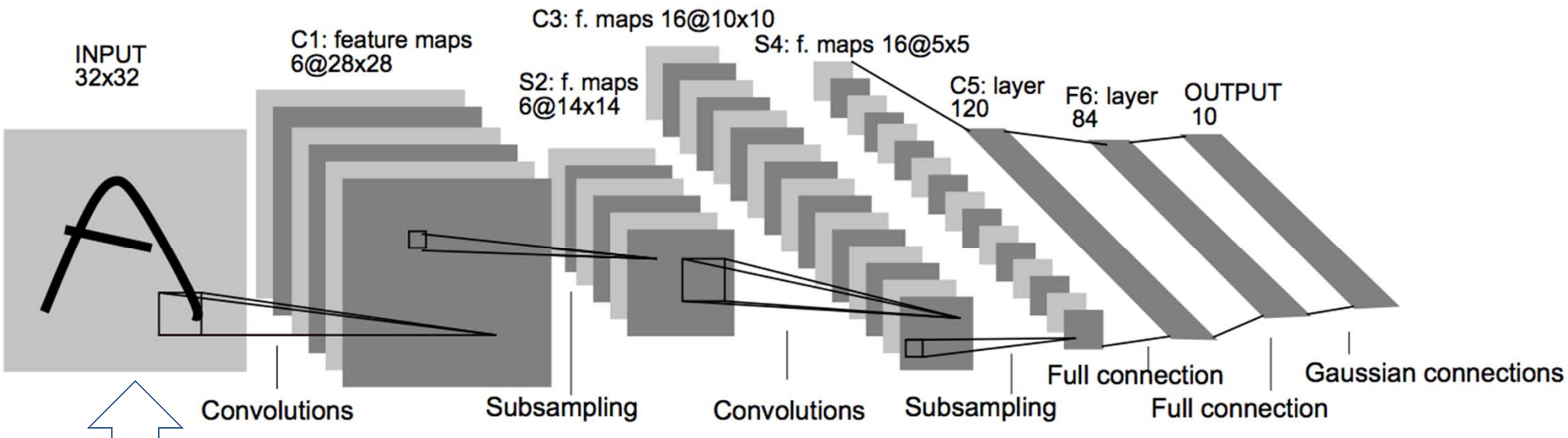


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

```
# import the necessary packages
from keras.datasets import mnist
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

LeNet5: Convolution

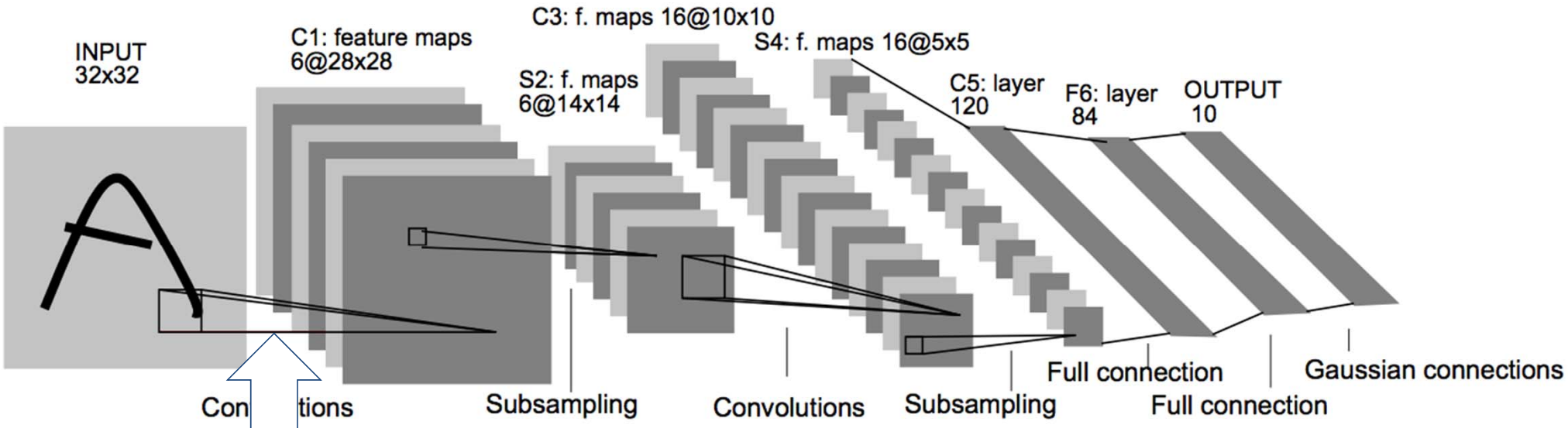


Fig. 2. Architecture o Net-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are trained to be identical.

```
model.add(Convolution2D(6, 5, 5,
                        border_mode="valid",
                        input_shape=(depth, height, width)))
```

LeNet5: Non-Linearity

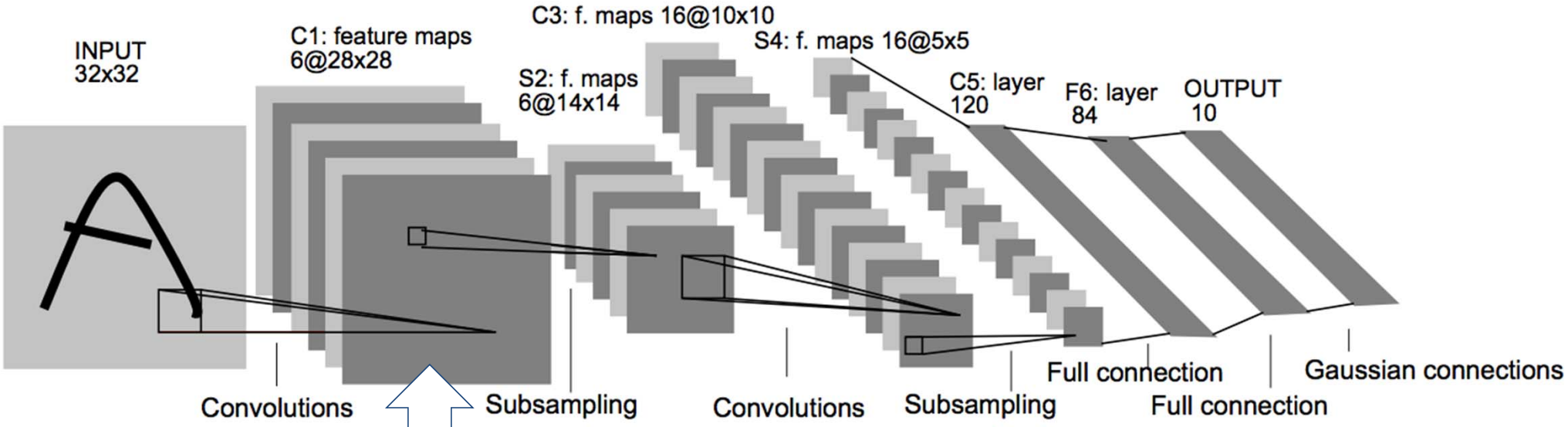


Fig. 2. Architecture of LeNet-5, a C olutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained tc identical.

```
from keras.layers.core import Activation
model.add(Activation(" sigmoid "))
```

LeNet5: Pooling

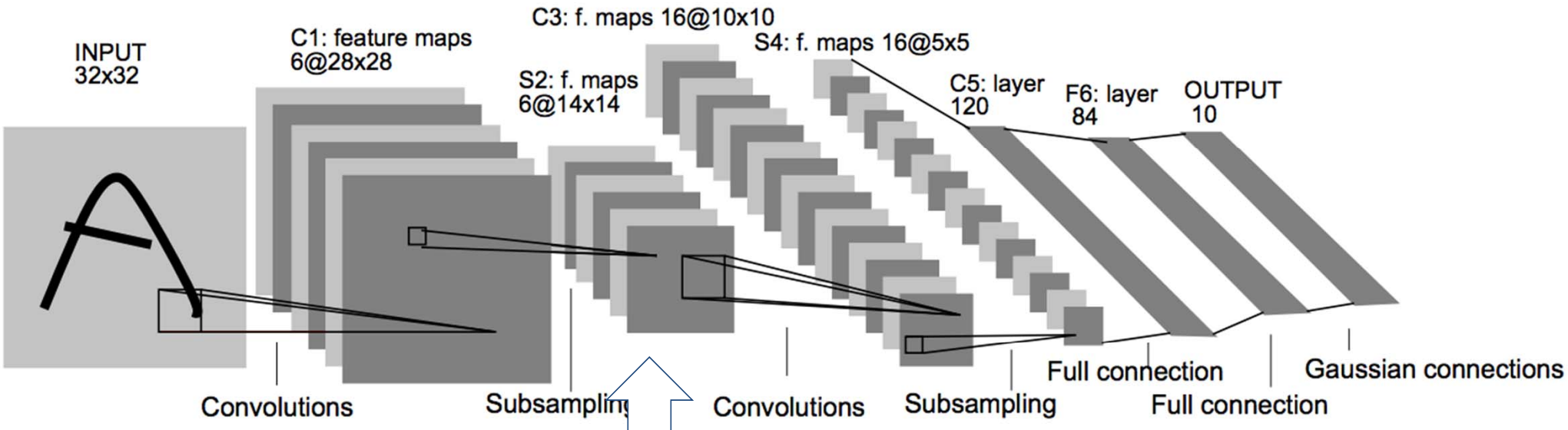


Fig. 2. Architecture of LeNet-5, a Convolutional Neural twork, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

```
from keras.layers.convolutional import MaxPooling2D
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
```

LeNet5: Second convolution

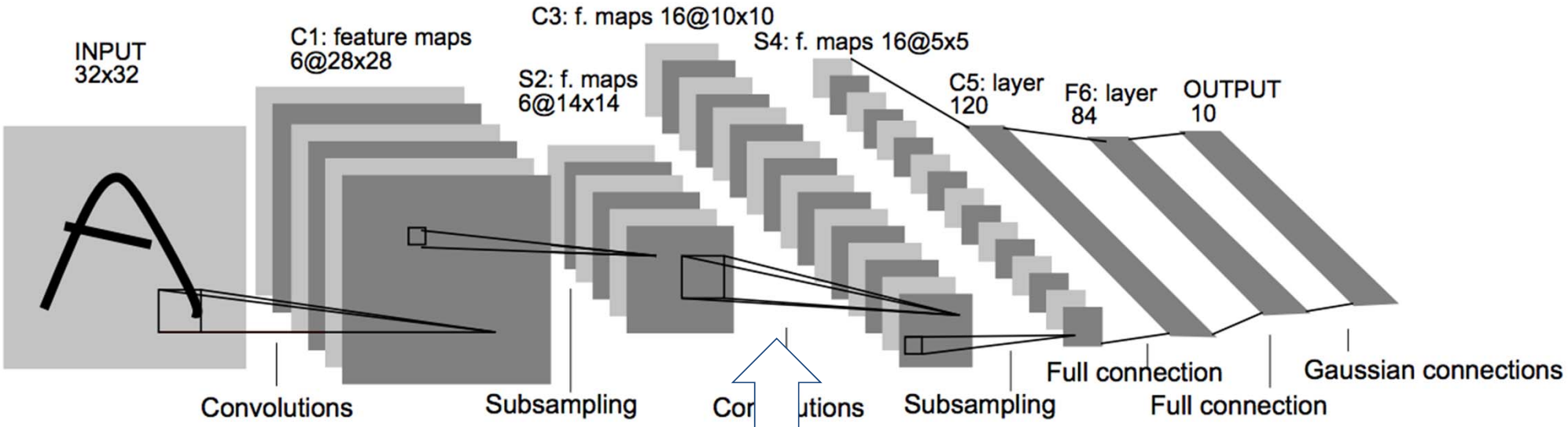


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

```
model.add(Convolution2D(16, 5, 5,
                        border_mode="valid"))
```

LeNet5: Second NonLinearity

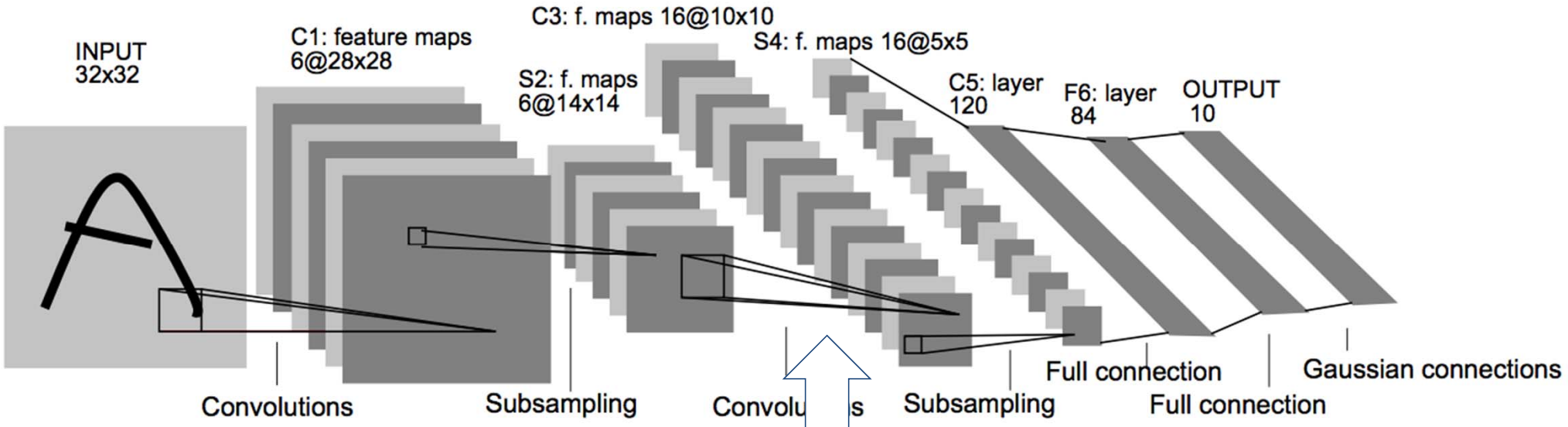


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

model.add(Activation(”sigmoid"))

LeNet5: Second Pooling

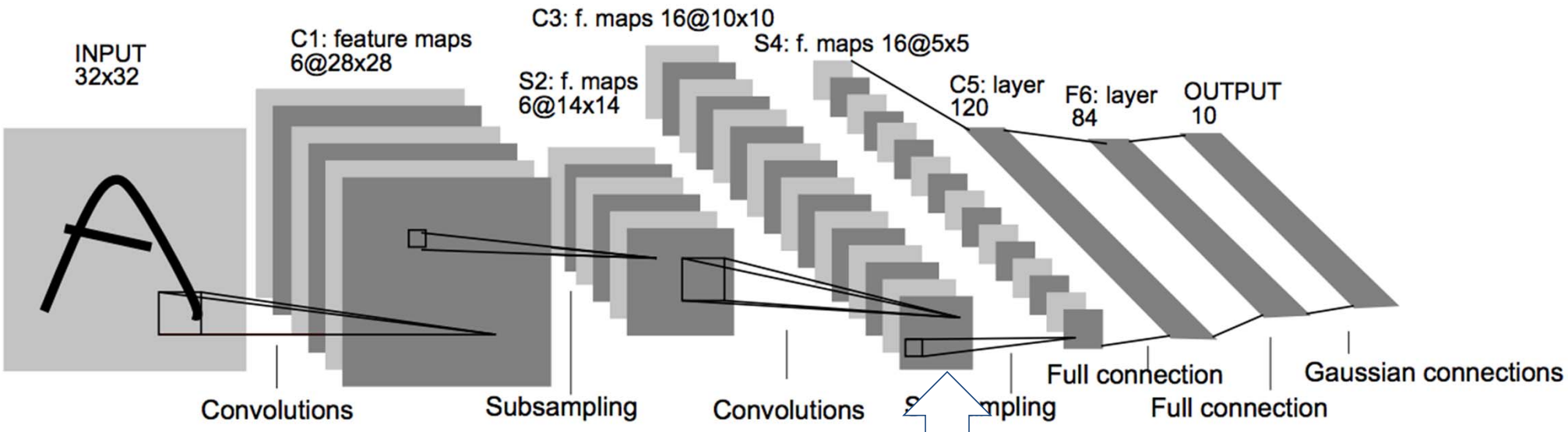


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits r nition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))

LeNet5: Third Convolution

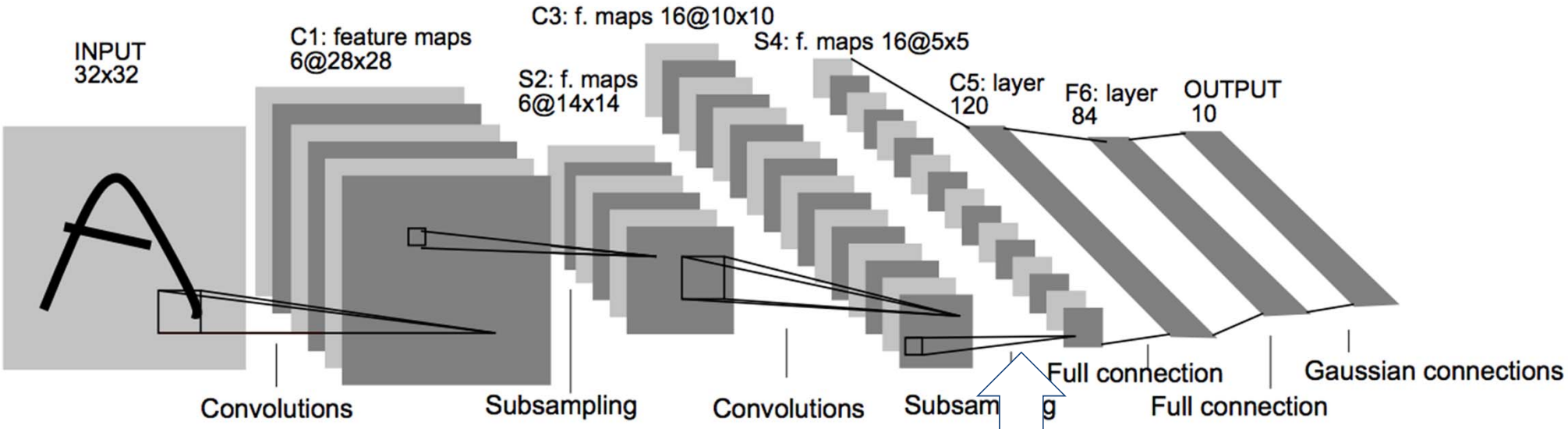


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recogni Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

model.add(Convolution2D(120, 1, 1, border_mode='valid'))

LeNet5: Flattening

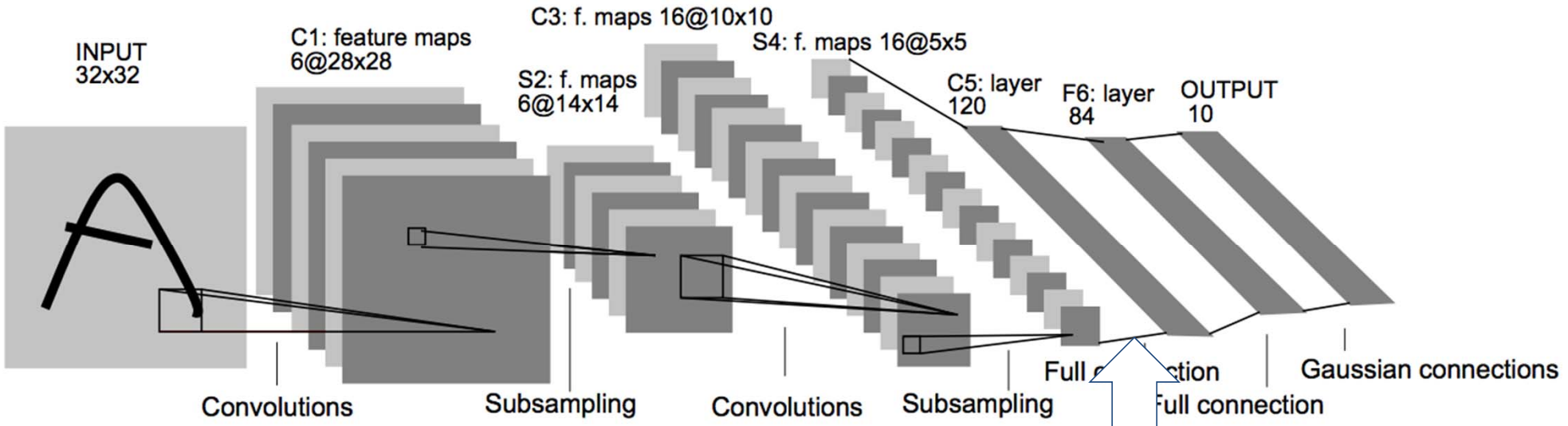


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

```
from keras.layers.core import Flatten
model.add(Flatten())
```

LeNet5: Dense Connections

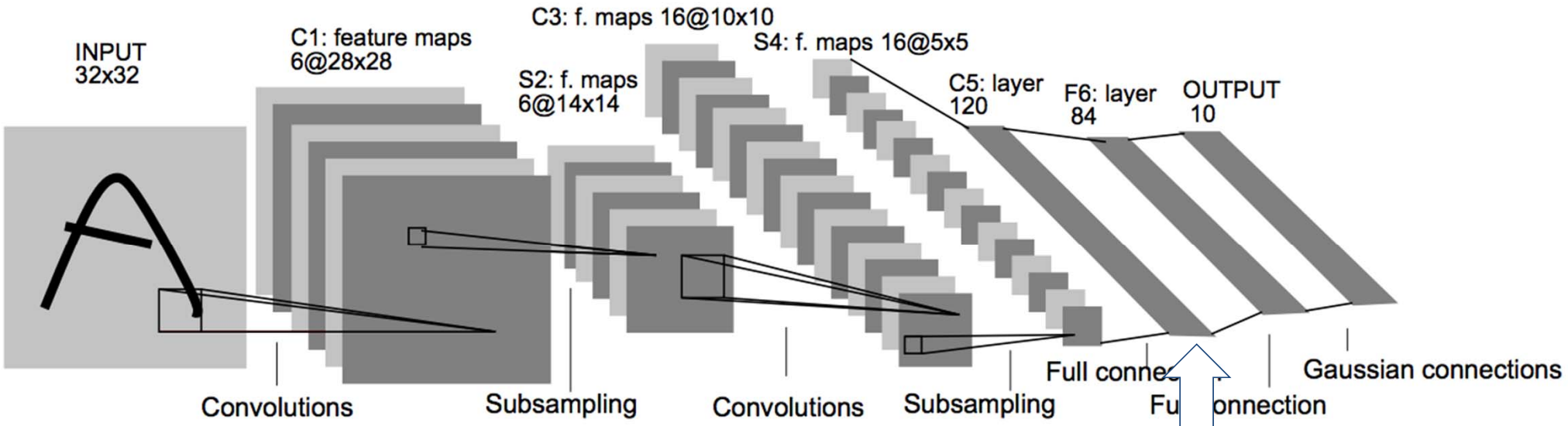


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

```
from keras.layers.core import Dense
model.add(Dense(84))
model.add(Activation("sigmoid"))
```

LeNet5: Dense Connections

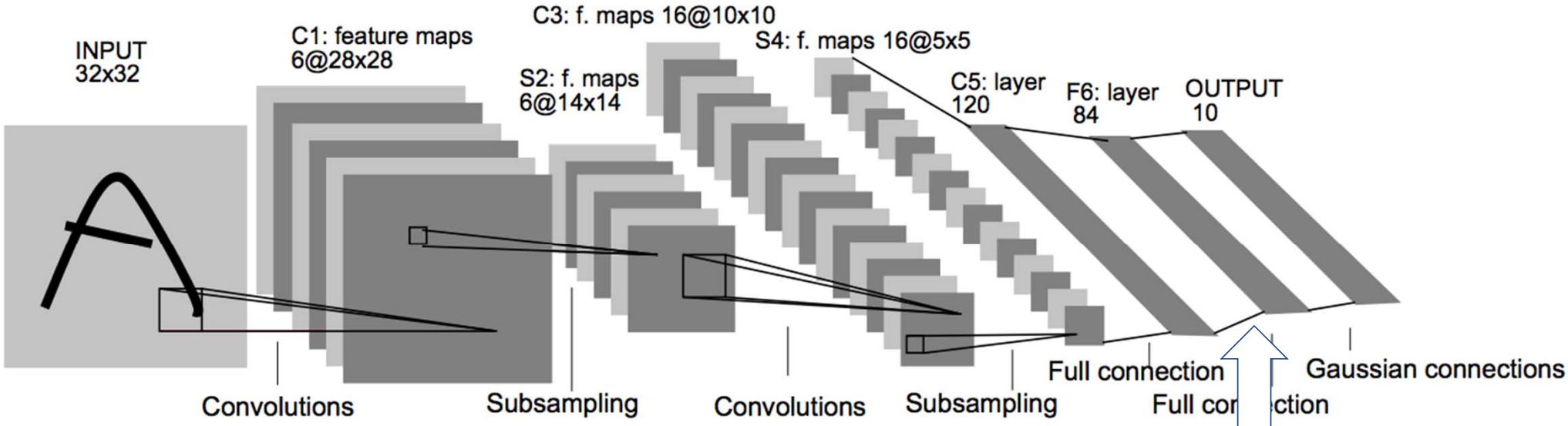


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a ture map, i.e. a set of units whose weights are constrained to be identical.

model.add(Dense(10))

LeNet5: Dense Connections

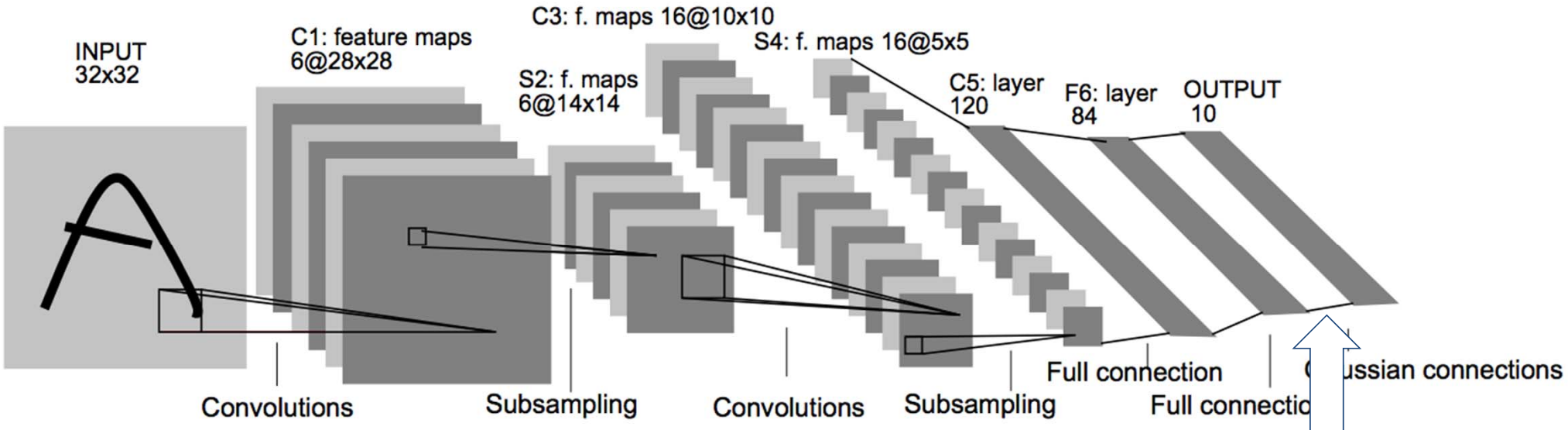


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature ap, i.e. a set of units whose weights are constrained to be identical.

model.add(Activation('softmax'))

ResNet

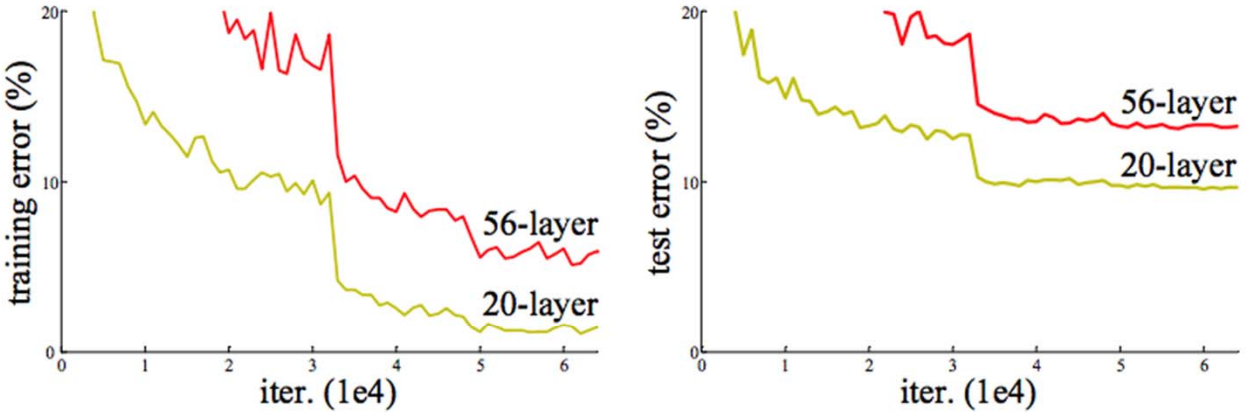


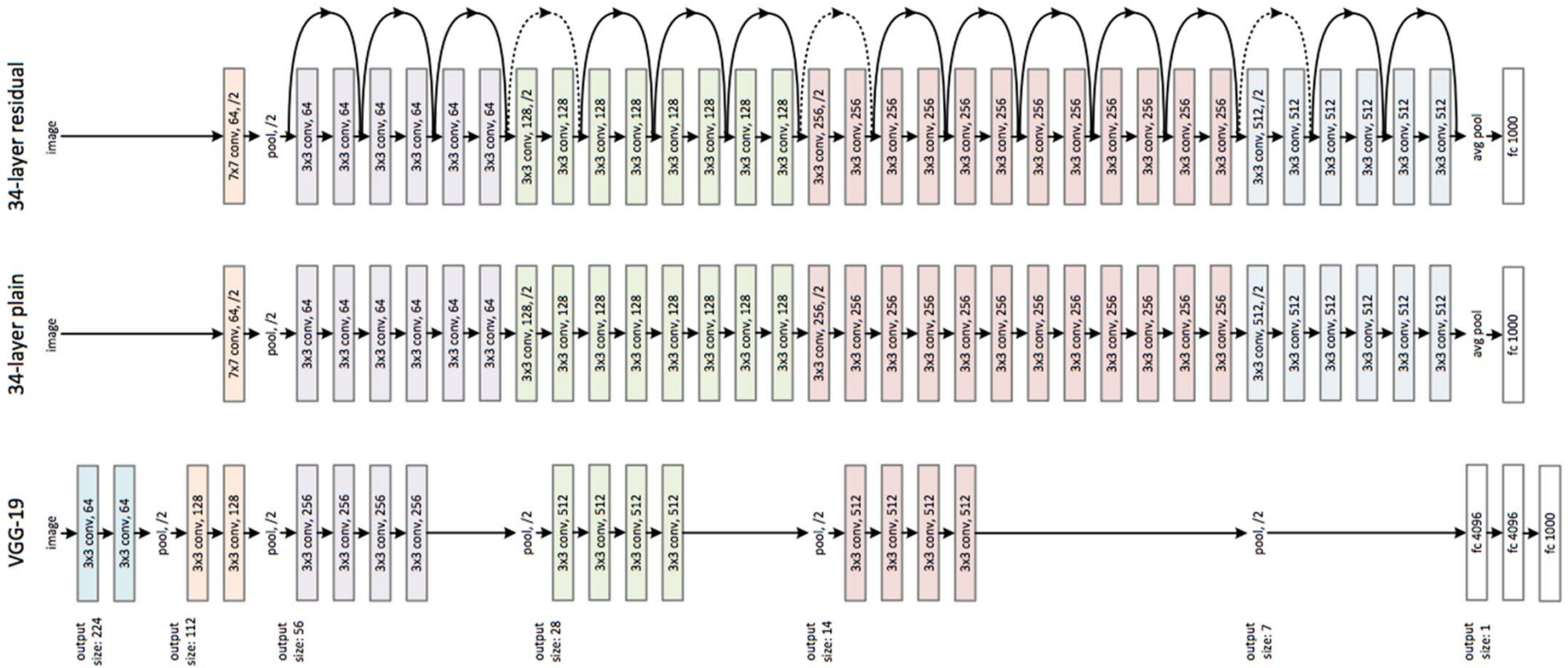
Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer "plain" networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

https://arxiv.org/pdf/1512.03385.pdf

- How can the behavior in the above charts be explained?
  - Vanishing gradients?
  - Overfitting?
  - Representation power?

ResNet Representation Power

https://arxiv.org/pdf/1512.03385.pdf



ResNet: Residual Learning Building Block

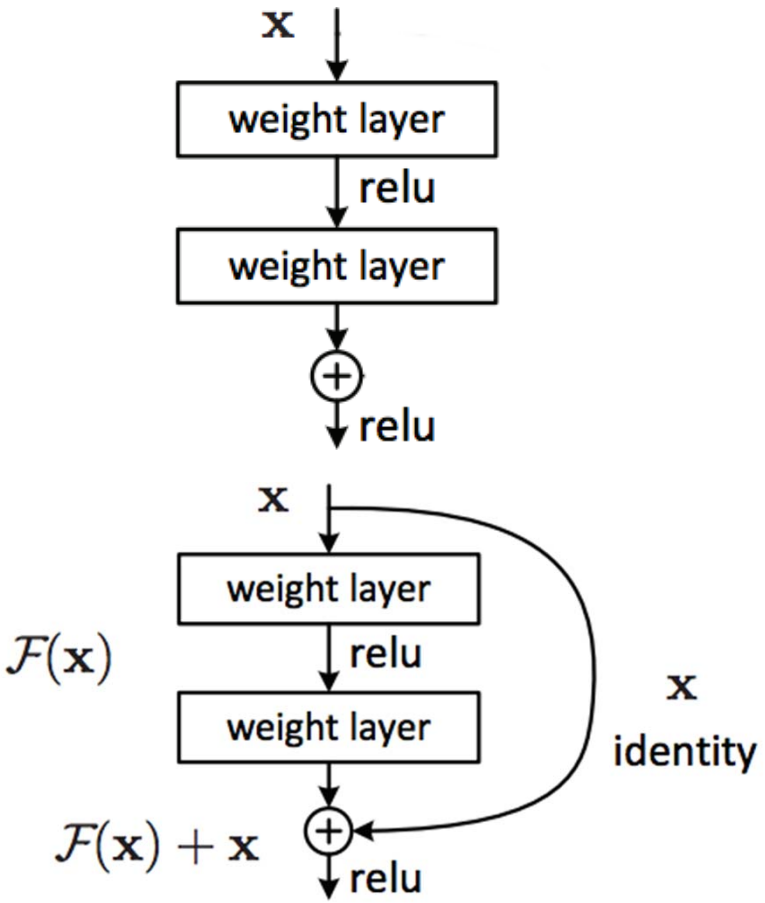
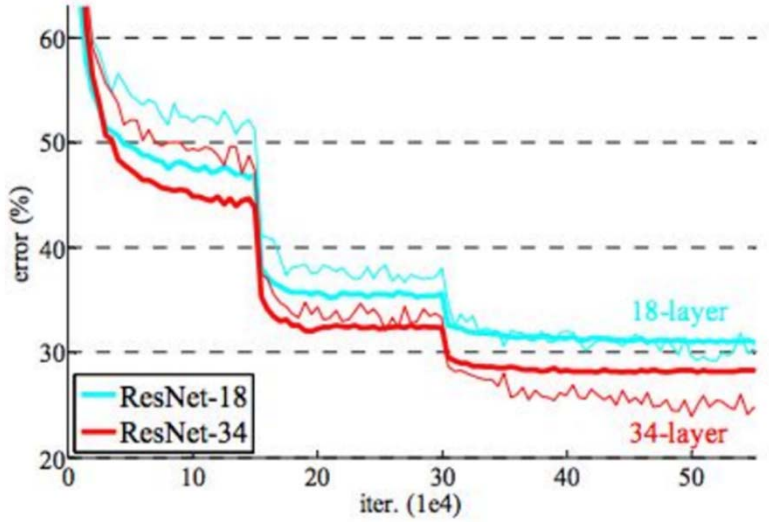


Figure 2. Residual learning: a building block.



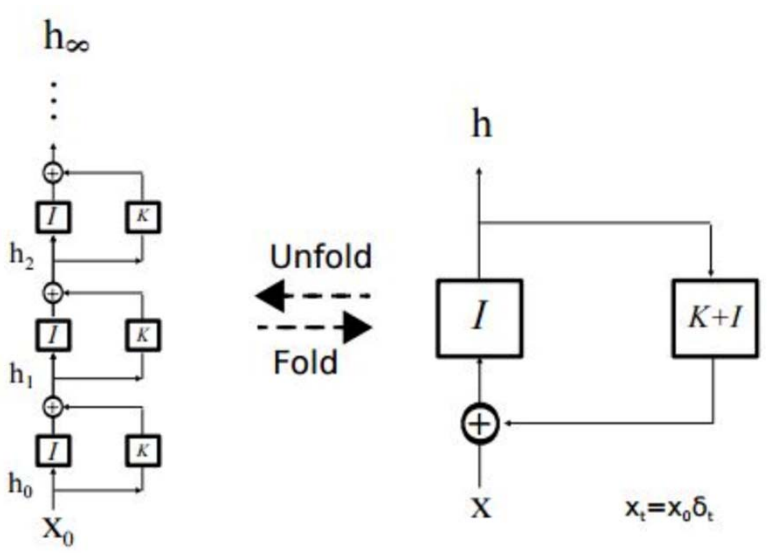
With residuals, the 34-layer network outperforms the 18 layer.

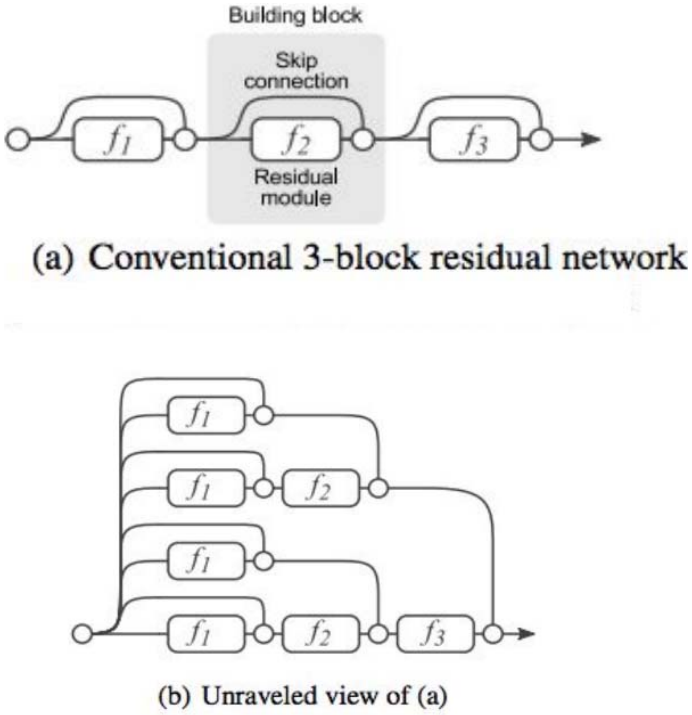
0	0	0
0	1	0
0	0	0

With 'SAME' padding, this will output the same feature map it receives as input

Why do ResNets work?

- Resembles ensembling shallower networks
- Can model recurrent computations
- Learning unrolled iterative estimations





Qianli Liao, Tomaso Poggio,Bridging the Gaps Between Residual Learning, Recurrent Neural Networks and Visual Cortex,