2-Feed-Forward Networks [RECAP backprop]

Lluis Guasch

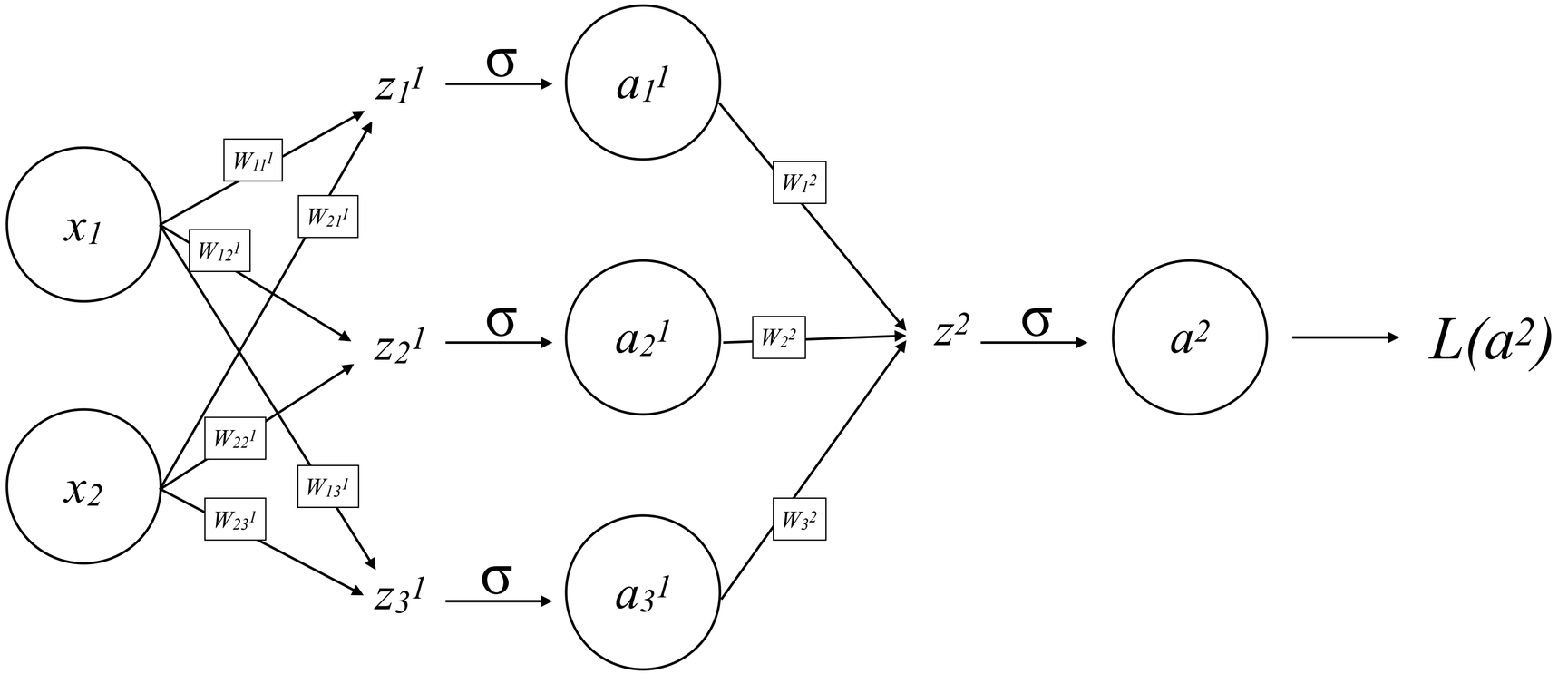
GTAs online:
Raul
Alex

Simple Network

We are going to go over the operations that give us the gradients for the parameters W^1 and W^2 we coded on Friday.

We will do this without bias terms, but extensions to cases with bias terms and different number of layers and neurons per layer follow the exact same mechanics (but keep in mind that the sizes of the matrices will obviously change).

Notation:



Elements of our network:

a^1 3x1

a^2 1x1

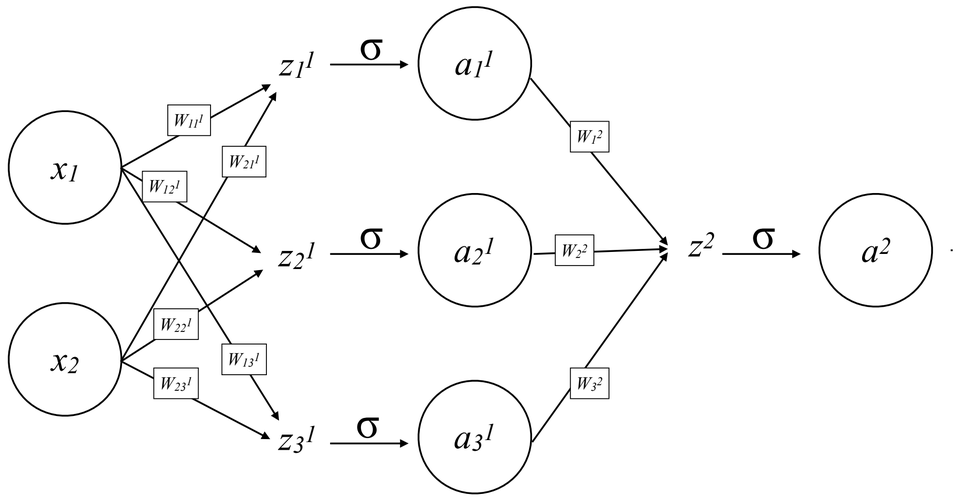
z^1 3x1

z^2 1x1

x 2x1

W^1 2x3 $\longrightarrow$ $W^1 = \begin{pmatrix} W_{11}^1 & W_{21}^1 & W_{31}^1 \\ W_{12}^1 & W_{22}^1 & W_{32}^1 \end{pmatrix}$

W^2 3x1



Simple Network

Elements of our network:

a^1 3x1

a^2 1x1

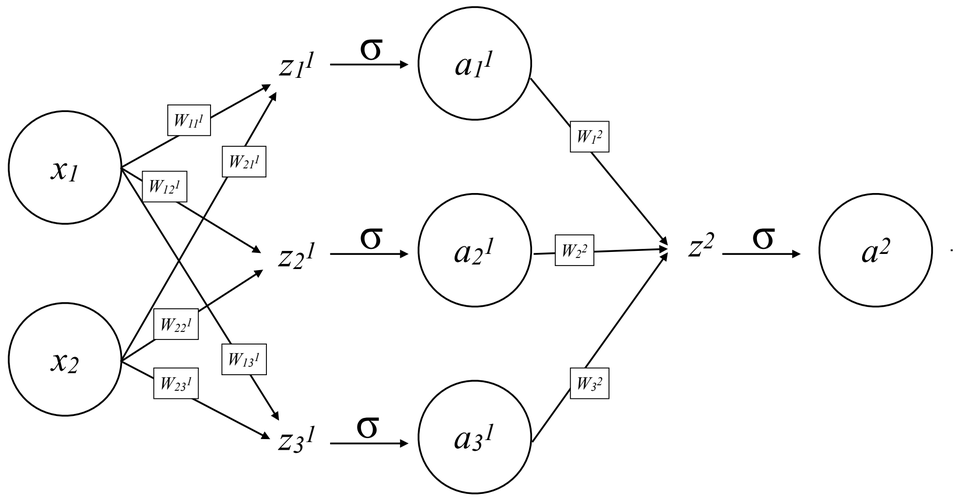
z^1 3x1

z^2 1x1

x 2x1

W^1 2x3 $\longrightarrow$ $W^1 = \begin{pmatrix} W^1_{11} & W^1_{21} & W^1_{31} \\ W^1_{12} & W^1_{22} & W^1_{32} \end{pmatrix}$

W^2 3x1



is it a good idea to pose the problem in matrix form?

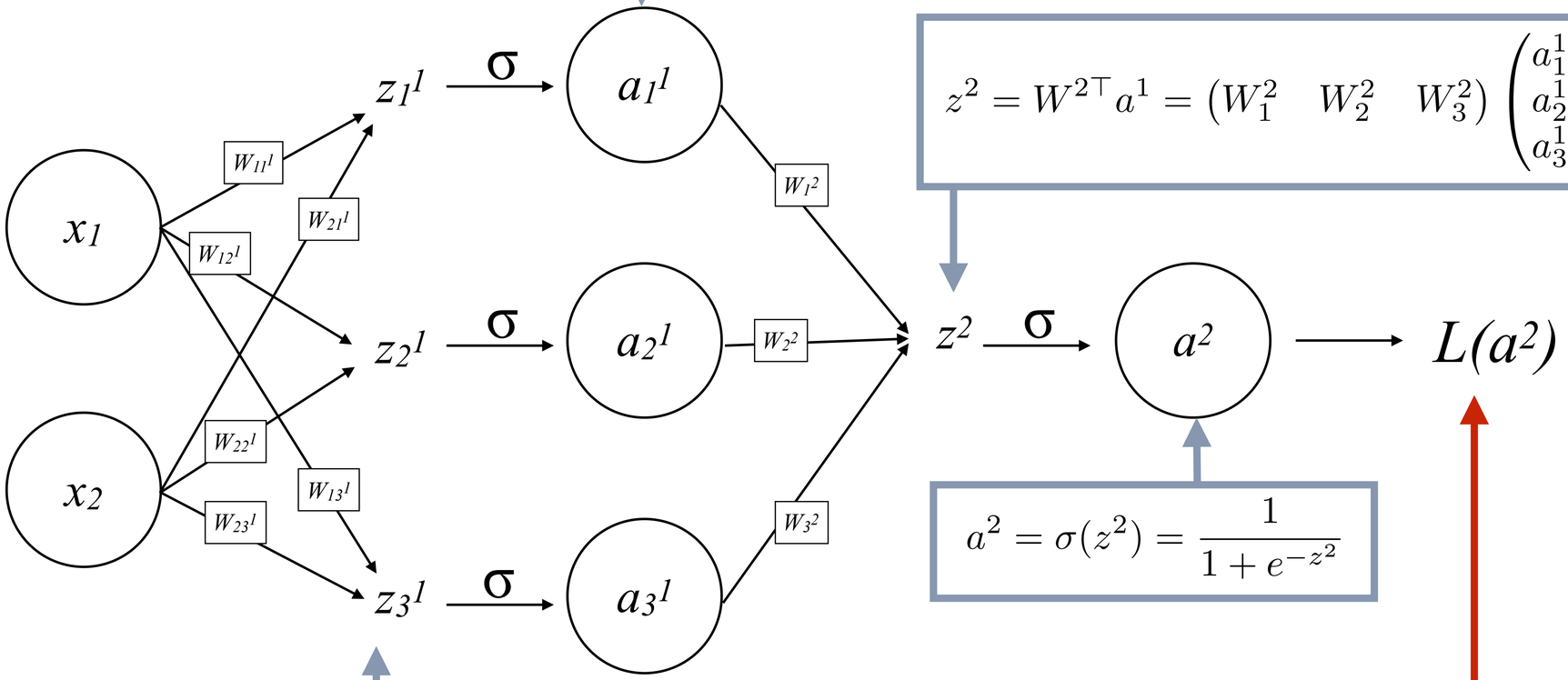
Consistency is the name of the game

Simple Network

Forward pass:

$$a^1 = \sigma(z^1) = \begin{pmatrix} \frac{1}{1+e^{-z_1^1}} \\ \frac{1}{1+e^{-z_2^1}} \\ \frac{1}{1+e^{-z_3^1}} \end{pmatrix} = \begin{pmatrix} a_1^1 \\ a_2^1 \\ a_3^1 \end{pmatrix}$$

$$z^2 = W^{2\top} a^1 = \begin{pmatrix} W_1^2 & W_2^2 & W_3^2 \end{pmatrix} \begin{pmatrix} a_1^1 \\ a_2^1 \\ a_3^1 \end{pmatrix}$$



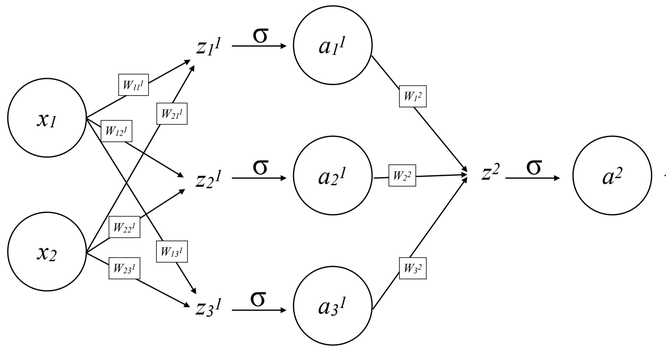
$$a^2 = \sigma(z^2) = \frac{1}{1+e^{-z^2}}$$

$$z^1 = W^{1\top} x = \begin{pmatrix} W_{11}^1 & W_{12}^1 \\ W_{21}^1 & W_{22}^1 \\ W_{31}^1 & W_{32}^1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} z_1^1 \\ z_2^1 \\ z_3^1 \end{pmatrix}$$

$$L(a^2) = -ylog(a^2) - (1-y)log(1-a^2)$$

Simple Network

We want to calculate the gradient for W^1 and for W^2:



$$\frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial W^2}$$

Most rapid changes in L with respect to W^2.

This is a vector with 3 entries, each one refers to a different value of W^2

$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial a^1}\frac{\partial a^1}{\partial z^1}\frac{\partial z^1}{\partial W^1}$$

Most rapid changes in L with respect to W^1.

This is a matrix with 3x2 entries, each one refers to a different value of W^2

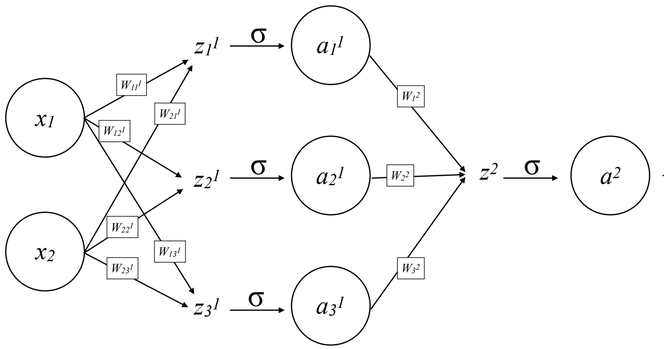
Simple Network

Let's start with W^2:

$$\frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial W^2}$$

□ x

$$\frac{\partial L}{\partial a^2} = \frac{(a^2 - y)}{(a^2(1 - a^2))} \longrightarrow$$ □ 1x1



Simple Network

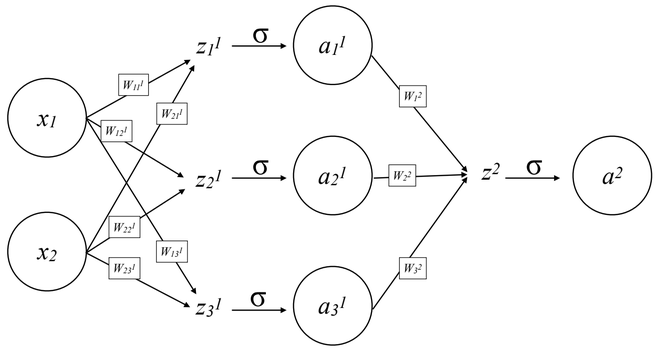
Let's start with W^2:

$$\frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial W^2}$$

□ x □ x

$$\frac{\partial L}{\partial a^2} = \frac{(a^2 - y)}{(a^2(1 - a^2))} \longrightarrow \square \text{ 1x1}$$

$$\frac{\partial a^2}{\partial z^2} = \sigma(z^2)(1 - \sigma(z^2)) = a^2(1 - a^2) \longrightarrow \square \text{ 1x1}$$



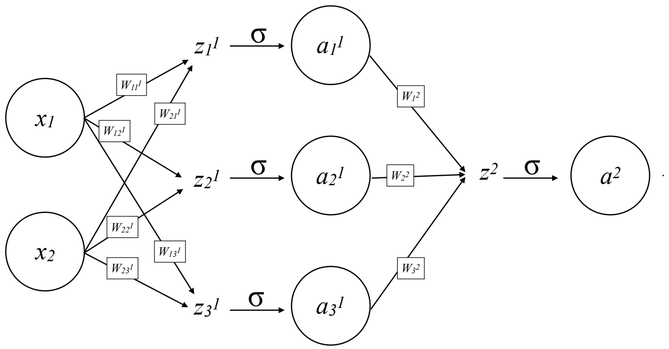
Simple Network

Let's start with W^2:

$$\frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial W^2}$$

$$\frac{\partial L}{\partial W^2}$$

□ x □ x ▯ = ▯ 1x3



$$\frac{\partial L}{\partial a^2} = \frac{(a^2 - y)}{(a^2(1 - a^2))} \longrightarrow$$ □ 1x1

$$\frac{\partial a^2}{\partial z^2} = \sigma(z^2)(1 - \sigma(z^2)) = a^2(1 - a^2) \longrightarrow$$ □ 1x1

$$\frac{\partial z^2}{\partial W^2} = \frac{\partial (W^{2\top} a^1)}{\partial W^2} = a^1 \longrightarrow$$ ▯ 1x3

denominator layout

a is not a function of **x**	$\frac{\partial(\mathbf{a}\cdot\mathbf{x})}{\partial \mathbf{x}} = \frac{\partial(\mathbf{x}\cdot\mathbf{a})}{\partial \mathbf{x}} =$ $\frac{\partial \mathbf{a}^\top \mathbf{x}}{\partial \mathbf{x}} = \frac{\partial \mathbf{x}^\top \mathbf{a}}{\partial \mathbf{x}} =$	$\mathbf{a}^\top$	$\mathbf{a}$

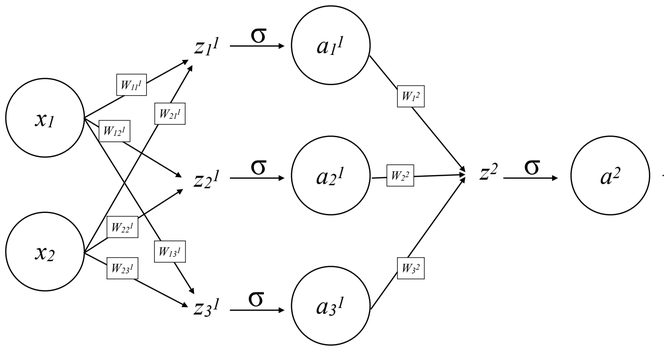
https://en.wikipedia.org/wiki/Matrix_calculus

Simple Network

Let's start with W^2:

$$\frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial W^2}$$

$$\frac{\partial L}{\partial W^2} = \square \times \square \times \begin{bmatrix}\square\end{bmatrix} = \begin{bmatrix}\square\end{bmatrix}$$ 1x3



$$\frac{\partial L}{\partial a^2} = \frac{(a^2 - y)}{(a^2(1-a^2))} \longrightarrow \square$$ 1x1

3 gradients, one for each parameter W^2_j

$$\frac{\partial a^2}{\partial z^2} = \sigma(z^2)(1-\sigma(z^2)) = a^2(1-a^2) \longrightarrow \square$$ 1x1

$$\frac{\partial z^2}{\partial W^2} = \frac{\partial (W^{2\top} a^1)}{\partial W^2} = a^1 \longrightarrow \begin{bmatrix}\square\end{bmatrix}$$ 1x3

denominator layout

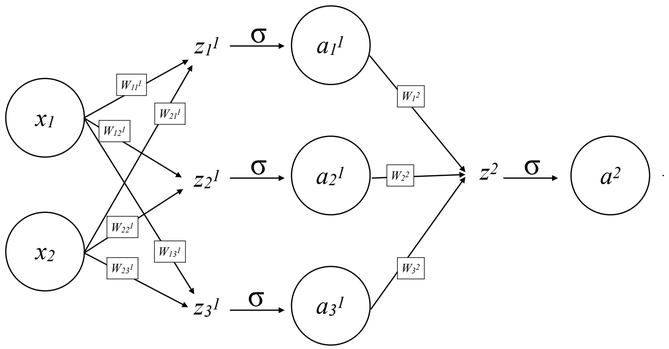
a is not a function of **x**	$\frac{\partial(\mathbf{a}\cdot\mathbf{x})}{\partial\mathbf{x}} = \frac{\partial(\mathbf{x}\cdot\mathbf{a})}{\partial\mathbf{x}} =$ $\frac{\partial\mathbf{a}^\top\mathbf{x}}{\partial\mathbf{x}} = \frac{\partial\mathbf{x}^\top\mathbf{a}}{\partial\mathbf{x}} =$	$\mathbf{a}^\top$	$\mathbf{a}$

https://en.wikipedia.org/wiki/Matrix_calculus

Simple Network

Matrix convention:

$$\frac{\partial z^2}{\partial W^2} = \frac{\partial (W^{2\top} a^1)}{\partial W^2} = a^1 \longrightarrow \begin{bmatrix} \\ \\ \end{bmatrix}$$ 3x1

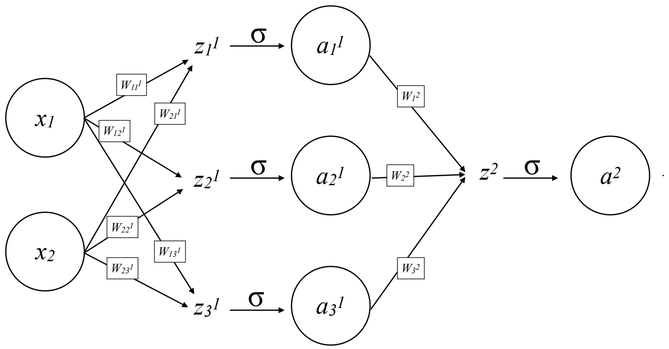


denominator layout

a is not a function of **x**	$\frac{\partial(\mathbf{a}\cdot\mathbf{x})}{\partial\mathbf{x}} = \frac{\partial(\mathbf{x}\cdot\mathbf{a})}{\partial\mathbf{x}} =$ $\frac{\partial\mathbf{a}^\top\mathbf{x}}{\partial\mathbf{x}} = \frac{\partial\mathbf{x}^\top\mathbf{a}}{\partial\mathbf{x}} =$	$\mathbf{a}^\top$	$\mathbf{a}$

https://en.wikipedia.org/wiki/Matrix_calculus

Simple Network



Now let's do W^1:

$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial a^1}\frac{\partial a^1}{\partial z^1}\frac{\partial z^1}{\partial W^1}$$

$$\frac{\partial L}{\partial a^2} = \frac{(a^2 - y)}{(a^2(1-a^2))} \longrightarrow \square \text{ 1x1}$$

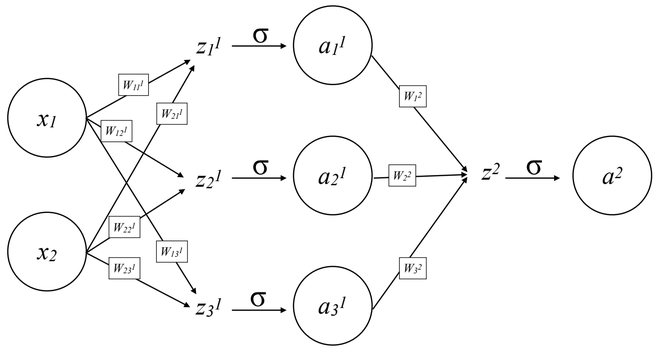
$$\frac{\partial a^2}{\partial z^2} = \sigma(z^2)(1-\sigma(z^2)) = a^2(1-a^2) \longrightarrow \square \text{ 1x1}$$

~~$$\frac{\partial z^2}{\partial W^2} = \frac{\partial W^{2\top} a^1}{\partial W^2} \longrightarrow \square \text{ 3x1}$$~~

Simple Network

Now let's do W^1:

$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial a^1}\frac{\partial a^1}{\partial z^1}\frac{\partial z^1}{\partial W^1}$$



$$\frac{\partial L}{\partial a^2} = \frac{(a^2 - y)}{(a^2(1 - a^2))} \longrightarrow \square \text{ 1x1}$$

$$\frac{\partial a^2}{\partial z^2} = \sigma(z^2)(1 - \sigma(z^2)) = a^2(1 - a^2) \longrightarrow \square \text{ 1x1}$$

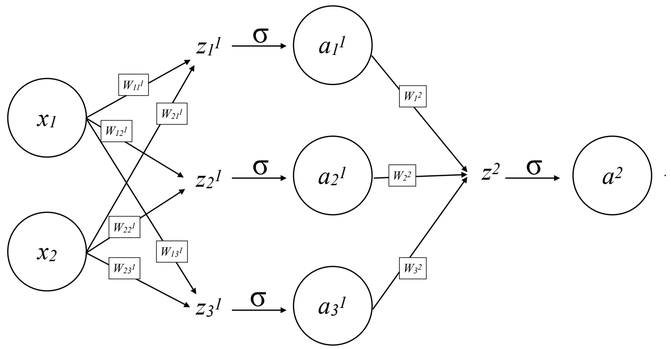
$$\frac{\partial z^2}{\partial a^1} = \frac{\partial W^{2\top} a^1}{\partial a^1} = W^2 \longrightarrow \square \text{ 1x3}$$

denominator layout

Simple Network

Now let's do W^1:

$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial a^1}\frac{\partial a^1}{\partial z^1}\frac{\partial z^1}{\partial W^1}$$



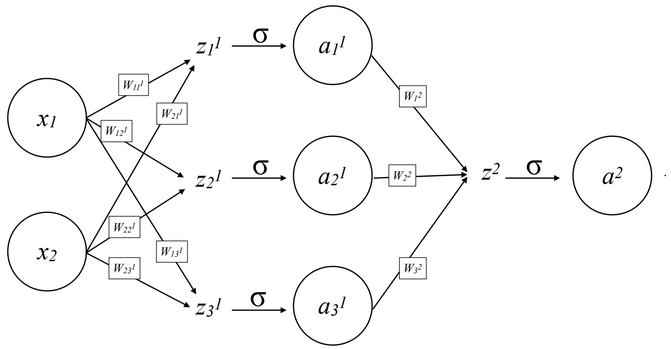
$$\frac{\partial a^1}{\partial z^1} = \begin{pmatrix} \frac{\partial a_1^1}{\partial z_1^1} & \frac{\partial a_1^1}{\partial z_2^1} & \frac{\partial a_1^1}{\partial z_2^1} \\ \frac{\partial a_2^1}{\partial z_1^1} & \frac{\partial a_2^1}{\partial z_2^1} & \frac{\partial a_2^1}{\partial z_2^1} \\ \frac{\partial a_3^1}{\partial z_1^1} & \frac{\partial a_3^1}{\partial z_2^1} & \frac{\partial a_3^1}{\partial z_2^1} \end{pmatrix} = \begin{pmatrix} a_1^1(1-a_1^1) & 0 & 0 \\ 0 & a_2^1(1-a_1^2) & \\ 0 & 0 & a_3^1(1-a_3^1) \end{pmatrix} \longrightarrow \square \quad 3\text{x}3$$

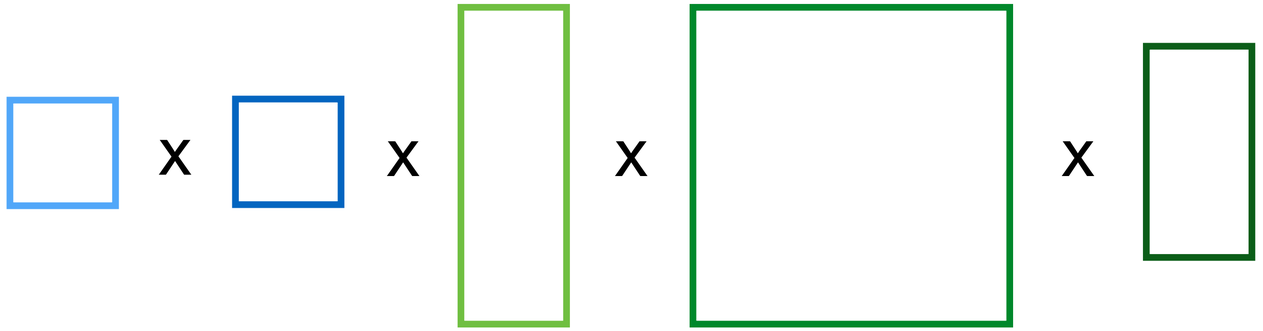
$$\frac{\partial z^1}{\partial W^1} = \frac{\partial (W^{1\top} x)}{\partial W^2} = x \longrightarrow \square \quad 1\text{x}2 \quad \text{denominator layout}$$

Simple Network

Putting it all together:

$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial a^1}\frac{\partial a^1}{\partial z^1}\frac{\partial z^1}{\partial W^1}$$

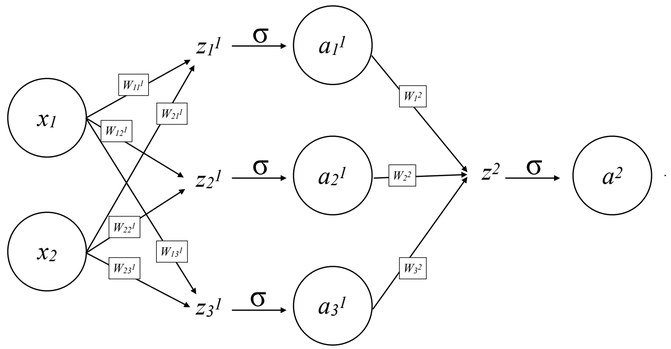


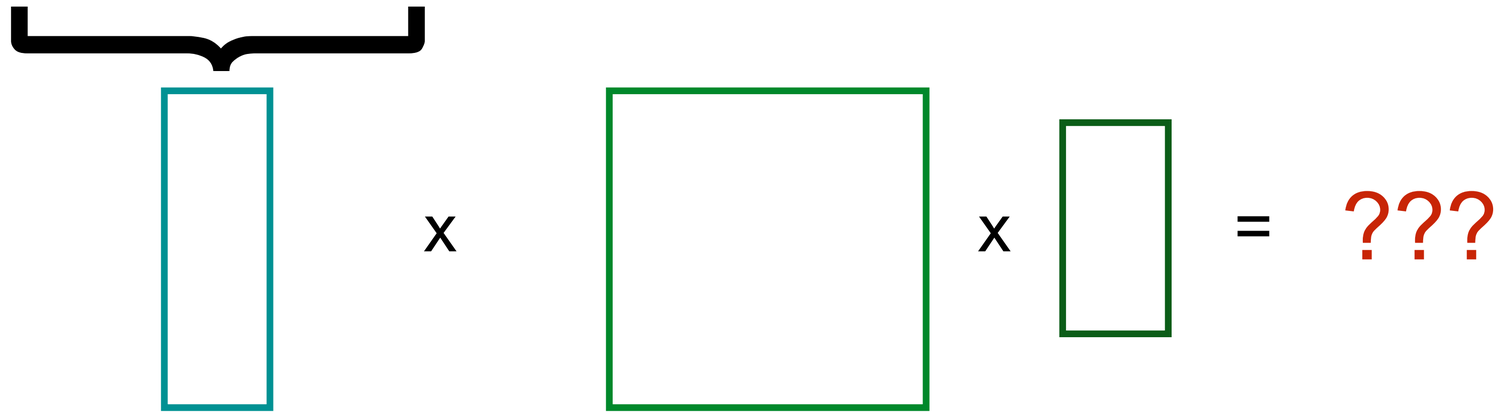


Simple Network

Putting it all together:

$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial a^1}\frac{\partial a^1}{\partial z^1}\frac{\partial z^1}{\partial W^1}$$

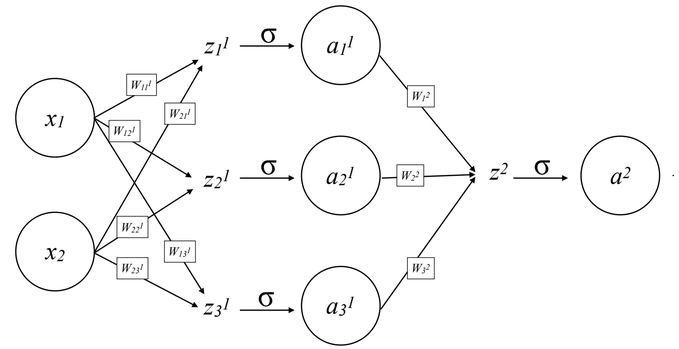


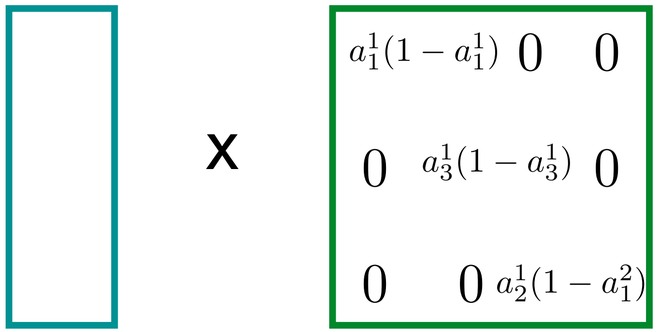


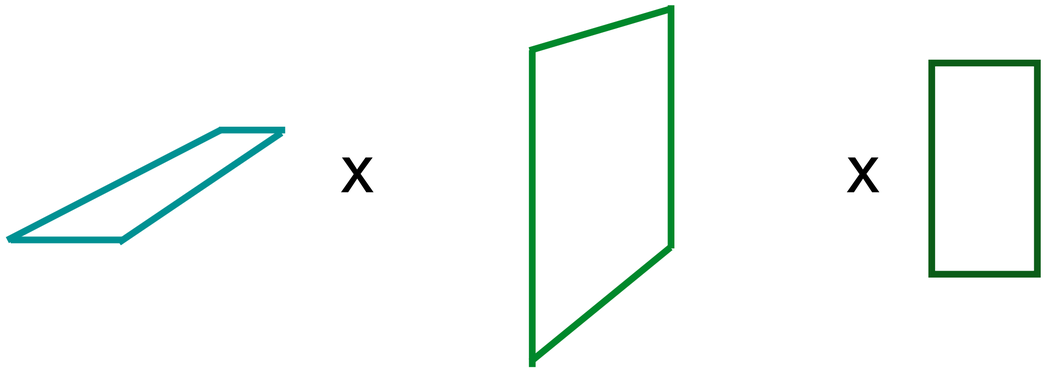
Simple Network

Putting it all together:

$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial a^1}\frac{\partial a^1}{\partial z^1}\frac{\partial z^1}{\partial W^1}$$



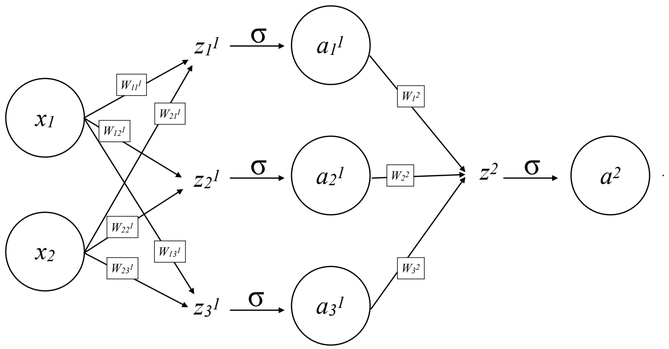


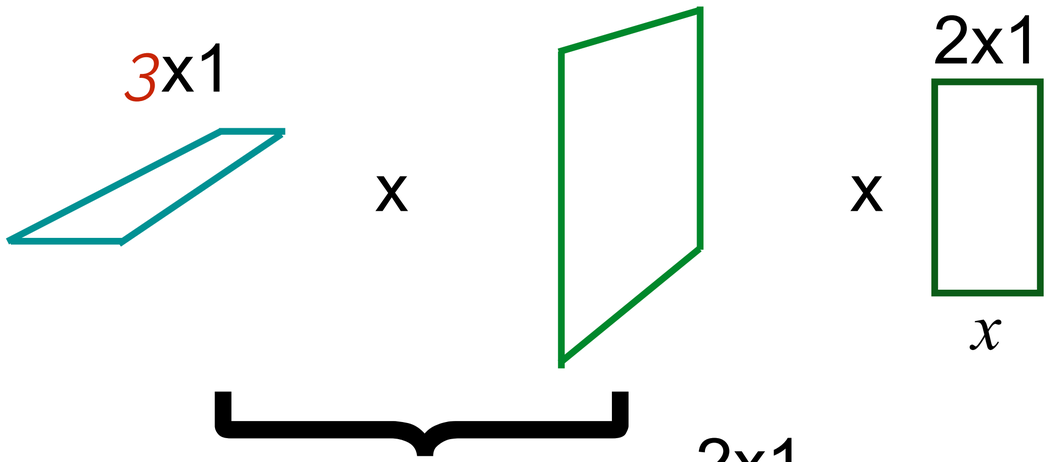


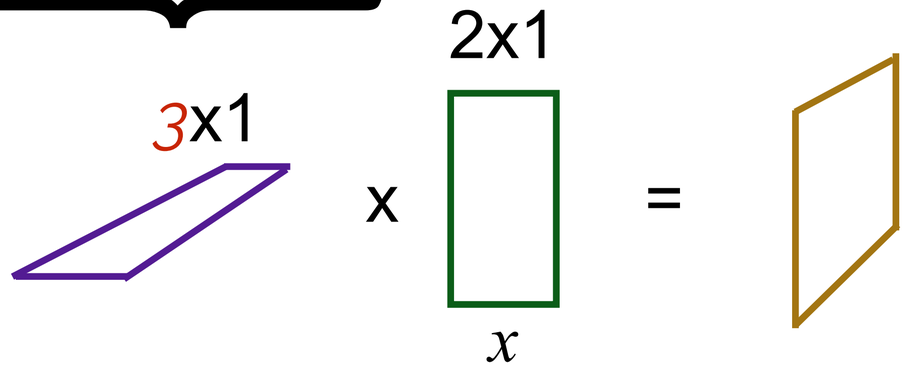
Simple Network

Putting it all together:

$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial a^1}\frac{\partial a^1}{\partial z^1}\frac{\partial z^1}{\partial W^1}$$





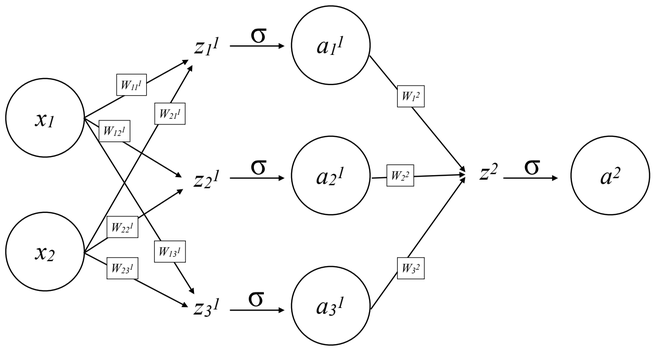


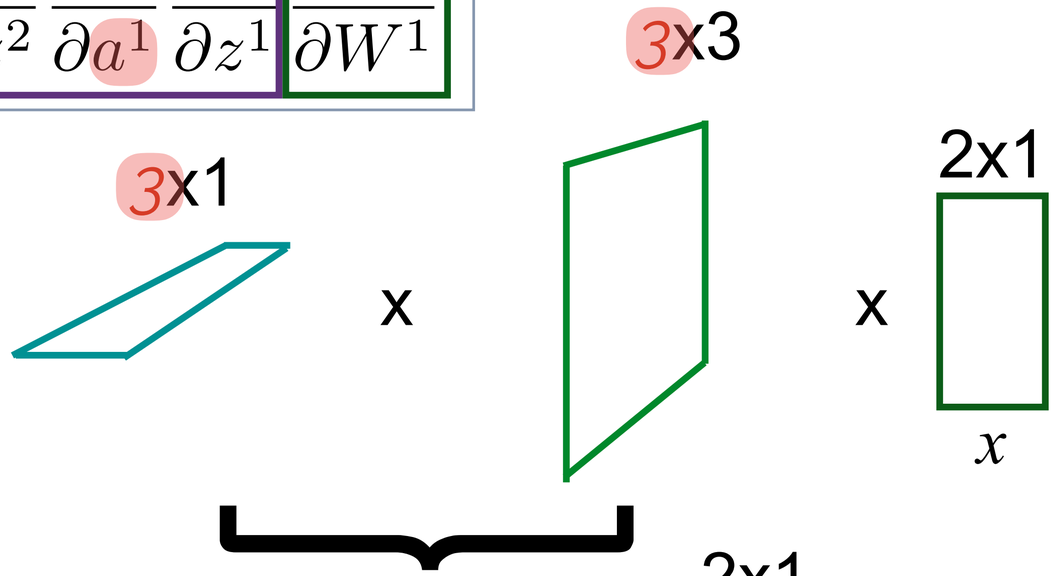
→ 6 gradients, one for each parameter W^1_{ij}

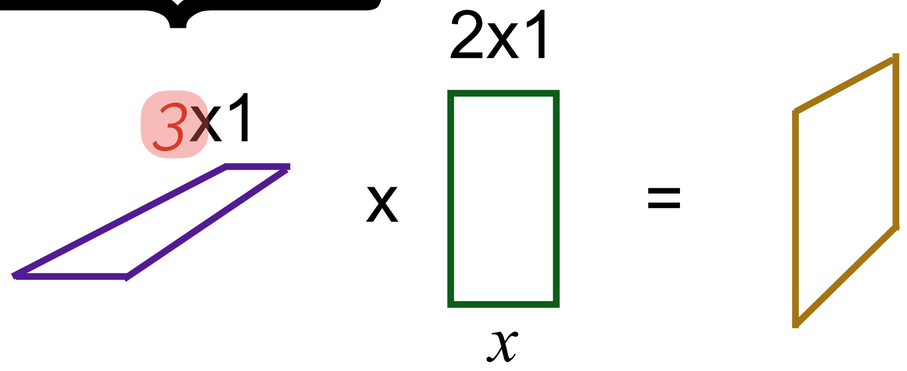
Simple Network

Putting it all together:

$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial a^1}\frac{\partial a^1}{\partial z^1}\frac{\partial z^1}{\partial W^1}$$







→ 6 gradients, one for each parameter W^1_{ij}

And now let's find how this relates to the codes we saw. **Forward**:

$$z^1 = W^{1\top} x = \begin{pmatrix} W^1_{11} & W^1_{12} \\ W^1_{21} & W^1_{22} \\ W^1_{31} & W^1_{32} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} z^1_1 \\ z^1_2 \\ z^1_3 \end{pmatrix}$$

$$a^1 = \sigma(z^1) = \begin{pmatrix} \frac{1}{1+e^{-z^1_1}} \\ \frac{1}{1+e^{-z^1_2}} \\ \frac{1}{1+e^{-z^1_3}} \end{pmatrix} = \begin{pmatrix} a^1_1 \\ a^1_2 \\ a^1_3 \end{pmatrix}$$

$$z^2 = W^{2\top} a^1 = \begin{pmatrix} W^2_1 & W^2_2 & W^2_3 \end{pmatrix} \begin{pmatrix} a^1_1 \\ a^1_2 \\ a^1_3 \end{pmatrix}$$

$$a^2 = \sigma(z^2) = \frac{1}{1+e^{-z^2}}$$

$$L(a^2) = -y log(a^2) - (1-y) log(1-a^2)$$

Python **manual** implementation:

```
#Forward Pass Layer 1
z1 = torch.matmul(X, W1)
a1 = sigmoid(z1)

#Forward Pass Layer 2
z2 =  torch.matmul(a1, W2)
a2 = sigmoid(z2)
```

PyTorch implementation:

```
class SingleHiddenLayerNetwork(nn.Module):
    def __init__(self, I, H, O):
        super(SingleHiddenLayerNetwork, self).__init__()
        self.hidden_1 = nn.Linear(I, H, bias=False)
        self.output = nn.Linear(H, O, bias=False)
        self.activation = nn.Sigmoid()

    def forward(self, X):
        z1 = self.hidden_1(X)
        a1 = self.activation(z1)
        z2 = self.output(a1)
        a2 = self.activation(z2)
        return a2
```

```
a2 = model(X)
```

And now let's find how this relates to the codes we saw. **Forward**:

$$z^1 = W^{1\top} x = \begin{pmatrix} W^1_{11} & W^1_{12} \\ W^1_{21} & W^1_{22} \\ W^1_{31} & W^1_{32} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} z^1_1 \\ z^1_2 \\ z^1_3 \end{pmatrix}$$

$$a^1 = \sigma(z^1) = \begin{pmatrix} \frac{1}{1+e^{-z^1_1}} \\ \frac{1}{1+e^{-z^1_2}} \\ \frac{1}{1+e^{-z^1_3}} \end{pmatrix} = \begin{pmatrix} a^1_1 \\ a^1_2 \\ a^1_3 \end{pmatrix}$$

$$z^2 = W^{2\top} a^1 = \begin{pmatrix} W^2_1 & W^2_2 & W^2_3 \end{pmatrix} \begin{pmatrix} a^1_1 \\ a^1_2 \\ a^1_3 \end{pmatrix}$$

$$a^2 = \sigma(z^2) = \frac{1}{1+e^{-z^2}}$$

$$L(a^2) = -y log(a^2) - (1-y) log(1-a^2)$$

Python **manual** implementation:

```
train_loss = -1./N*(y*torch.log(a2)+(1-y)*torch.log(1-a2)).sum(0)
```

PyTorch implementation:

```
def bce_loss(y, a2):
    return -1/y.size(0)*(y*a2.log()+(1-y)*(1-a2).log()).sum(0)
```

And now let's find how this relates to the codes we saw. **Backprop**:

$$\frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial W^2}$$

$$\frac{\partial L}{\partial a^2} = \frac{(a^2 - y)}{(a^2(1 - a^2))}$$

$$\frac{\partial a^2}{\partial z^2} = \sigma(z^2)(1 - \sigma(z^2)) = a^2(1 - a^2)$$

$$\frac{\partial z^2}{\partial W^2} = \frac{\partial (W^{2\top} a^1)}{\partial W^2} = a^1$$

Python **manual** implementation:

```
dL_da2 = (a2-y)/(a2*(1-a2))
da2_dz2 = dSigmoid(a2)

dL_dW2 = torch.matmul(torch.transpose(a1, 0, 1), dL_da2*da2_dz2)
```

PyTorch implementation:

```
loss.backward()
```

And now let's find how this relates to the codes we saw. **Backprop**:

$$\frac{\partial L}{\partial W^2} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial W^2}$$

$$\frac{\partial L}{\partial a^2} = \frac{(a^2 - y)}{(a^2(1-a^2))}$$

$$\frac{\partial a^2}{\partial z^2} = \sigma(z^2)(1-\sigma(z^2)) = a^2(1-a^2)$$

$$\frac{\partial z^2}{\partial W^2} = \frac{\partial (W^{2\top} a^1)}{\partial W^2} = a^1$$

Python **manual** implementation:

```
dL_da2 = (a2-y)/(a2*(1-a2))
da2_dz2 = dSigmoid(a2)

dL_dW2 = torch.matmul(torch.transpose(a1, 0, 1), dL_da2*da2_dz2)
```

- broadcasting operations over dataset (the tensors `dL_da2`, `da2_dz2`, etc, have an extra dimension of size 1000).
- the matmul stacks all the gradients of `dL_dW2` for all the 1000 datapoints.
- We transpose `a1` so that the dot product makes sense.

PyTorch implementation:

```
loss.backward()
```

And now let's find how this relates to the codes we saw. **Backprop**:

$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial a^1}\frac{\partial a^1}{\partial z^1}\frac{\partial z^1}{\partial W^1}$$

$$\frac{\partial z^2}{\partial a^1} = \frac{\partial W^{2\top} a^1}{\partial a^1} = W^2$$

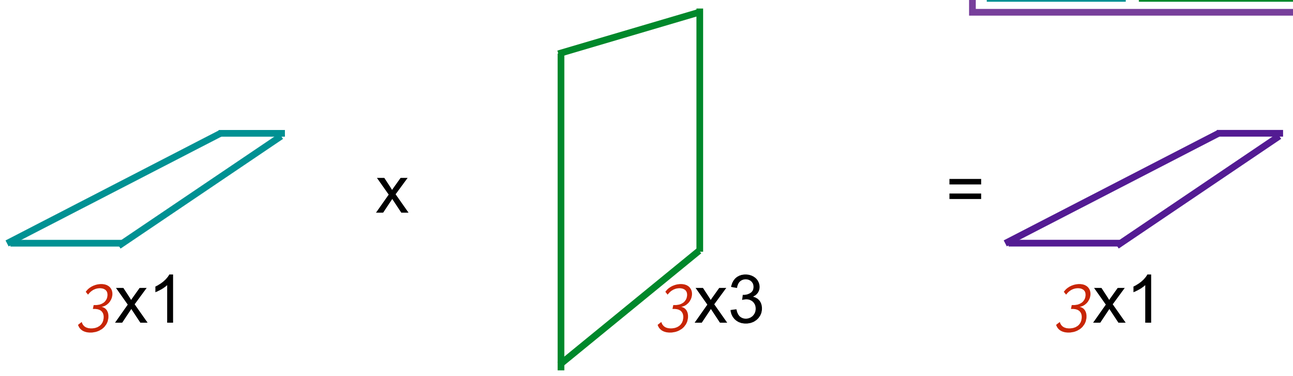
$$\frac{\partial a^1}{\partial z^1} = \begin{pmatrix} \frac{\partial a_1^1}{\partial z_1^1} & \frac{\partial a_1^1}{\partial z_2^1} & \frac{\partial a_1^1}{\partial z_3^1} \\ \frac{\partial a_2^1}{\partial z_1^1} & \frac{\partial a_2^1}{\partial z_2^1} & \frac{\partial a_2^1}{\partial z_3^1} \\ \frac{\partial a_3^1}{\partial z_1^1} & \frac{\partial a_3^1}{\partial z_2^1} & \frac{\partial a_3^1}{\partial z_3^1} \end{pmatrix} = \begin{pmatrix} a_1^1(1-a_1^1) & 0 & 0 \\ 0 & a_2^1(1-a_1^2) & \\ 0 & 0 & a_3^1(1-a_3^1) \end{pmatrix}$$

$$\frac{\partial z^1}{\partial W^1} = \frac{\partial (W^{1\top} x)}{\partial W^2} = x$$

Python **manual** implementation:

```
#Backward Pass Layer 1
dL_da1 = torch.matmul(dL_da2*da2_dz2, torch.transpose(W2, 0, 1))
da1_dz1 = dSigmoid(a1)

dL_dW1 = torch.matmul(torch.transpose(X, 0, 1), dL_da1*da1_dz1)
```



PyTorch implementation:

```
loss.backward()
```

Simple Network

And now let's find how this relates to the codes we saw. **Backprop**:

$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial a^1}\frac{\partial a^1}{\partial z^1}\frac{\partial z^1}{\partial W^1}$$

$$\frac{\partial z^2}{\partial a^1} = \frac{\partial W^{2\top} a^1}{\partial a^1} = W^2$$

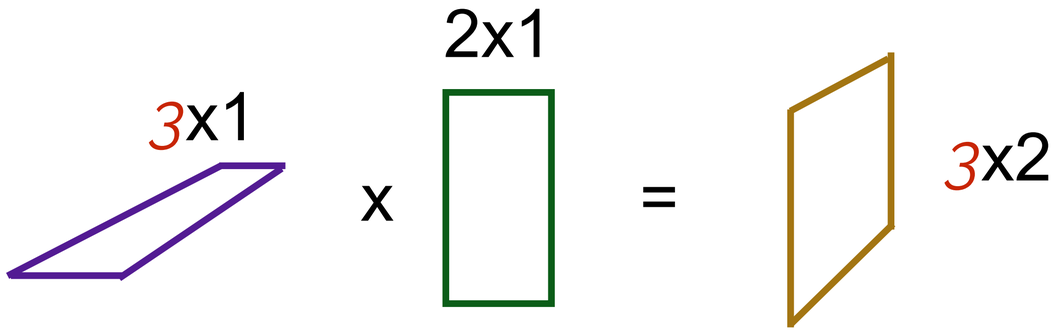
$$\frac{\partial a^1}{\partial z^1} = \begin{pmatrix} \frac{\partial a_1^1}{\partial z_1^1} & \frac{\partial a_1^1}{\partial z_2^1} & \frac{\partial a_1^1}{\partial z_2^1} \\ \frac{\partial a_2^1}{\partial z_1^1} & \frac{\partial a_2^1}{\partial z_2^1} & \frac{\partial a_2^1}{\partial z_2^1} \\ \frac{\partial a_3^1}{\partial z_1^1} & \frac{\partial a_3^1}{\partial z_2^1} & \frac{\partial a_3^1}{\partial z_2^1} \end{pmatrix} = \begin{pmatrix} a_1^1(1-a_1^1) & 0 & 0 \\ 0 & a_2^1(1-a_1^2) & \\ 0 & 0 & a_3^1(1-a_3^1) \end{pmatrix}$$

$$\frac{\partial z^1}{\partial W^1} = \frac{\partial (W^{1\top} x)}{\partial W^2} = x$$

Python **manual** implementation:

```
#Backward Pass Layer 1
dL_da1 = torch.matmul(dL_da2*da2_dz2, torch.transpose(W2, 0, 1))
da1_dz1 = dSigmoid(a1)

dL_dW1 = torch.matmul(torch.transpose(X, 0, 1), dL_da1*da1_dz1)
```



PyTorch implementation:

```
loss.backward()
```

And now let's find how this relates to the codes we saw. **Backprop**:

$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial a^2}\frac{\partial a^2}{\partial z^2}\frac{\partial z^2}{\partial a^1}\frac{\partial a^1}{\partial z^1}\frac{\partial z^1}{\partial W^1}$$

$$\frac{\partial z^2}{\partial a^1} = \frac{\partial W^{2\top} a^1}{\partial a^1} = W^2$$

$$\frac{\partial a^1}{\partial z^1} = \begin{pmatrix} \frac{\partial a_1^1}{\partial z_1^1} & \frac{\partial a_1^1}{\partial z_2^1} & \frac{\partial a_1^1}{\partial z_2^1} \\ \frac{\partial a_2^1}{\partial z_1^1} & \frac{\partial a_2^1}{\partial z_2^1} & \frac{\partial a_2^1}{\partial z_2^1} \\ \frac{\partial a_3^1}{\partial z_1^1} & \frac{\partial a_3^1}{\partial z_2^1} & \frac{\partial a_3^1}{\partial z_2^1} \end{pmatrix} = \begin{pmatrix} a_1^1(1-a_1^1) & 0 & 0 \\ 0 & a_2^1(1-a_1^2) & \\ 0 & 0 & a_3^1(1-a_3^1) \end{pmatrix}$$

$$\frac{\partial z^1}{\partial W^1} = \frac{\partial (W^{1\top} x)}{\partial W^2} = x$$

Python **manual** implementation:

```
#Backward Pass Layer 1
dL_da1 = torch.matmul(dL_da2*da2_dz2, torch.transpose(W2, 0, 1))
da1_dz1 = dSigmoid(a1)

dL_dW1 = torch.matmul(torch.transpose(X, 0, 1), dL_da1*da1_dz1)
```

- again, the transpose operations are to compute dot products along the dimension with 1000 elements. This sum over all gradients is then normalised when we applied the optimisation step:

```
W1 = W1 - lr/N*dL_dW1
W2 = W2 - lr/N*dL_dW2
```

PyTorch implementation:

```
loss.backward()
optimizer.step()
```

Summary:

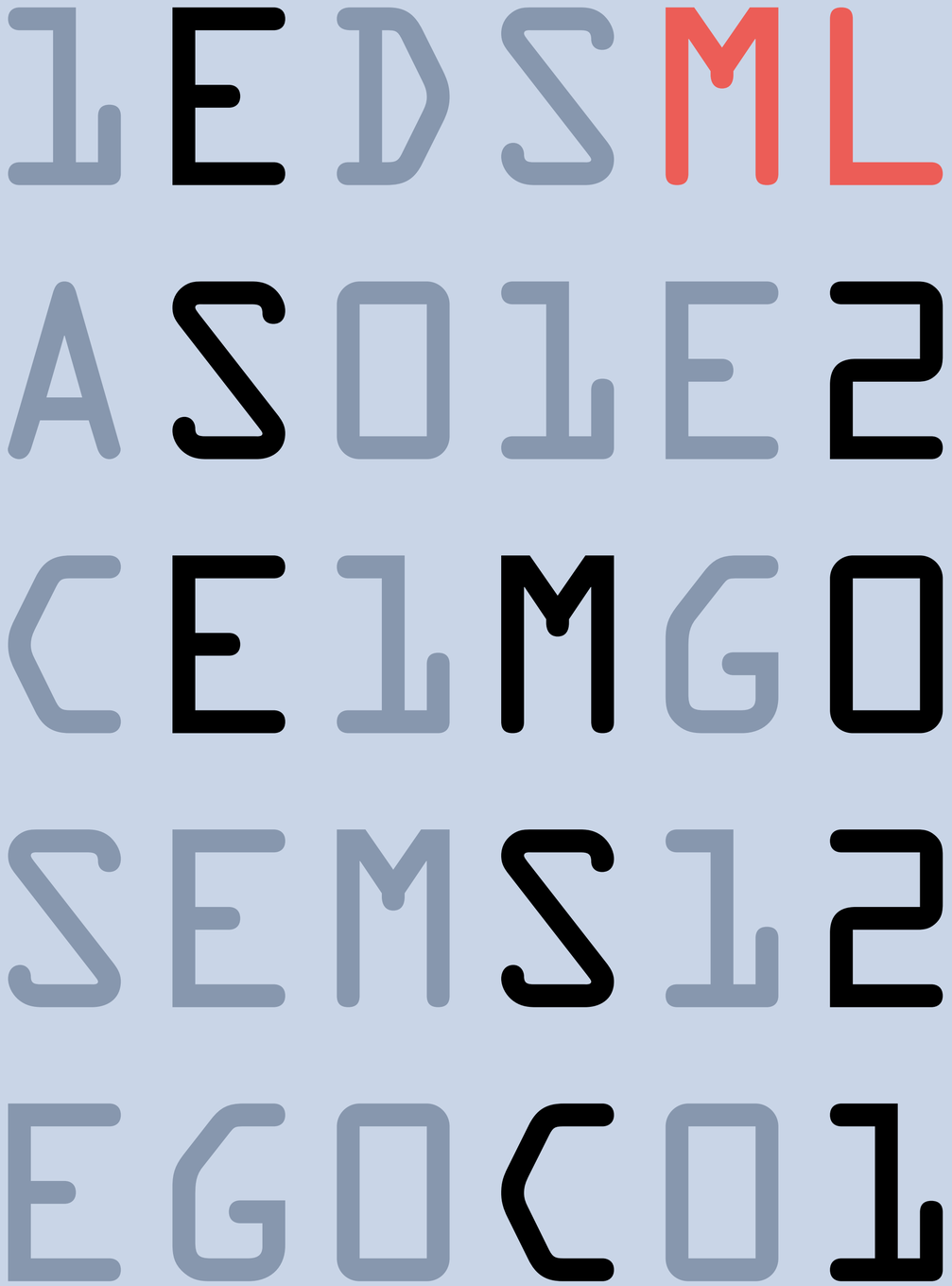
- Expressing backpropagation as a series of matrices containting derivative values is convenient to understand how to code it by hand, but makes it hard to do algebraic operations as they are really tensors and we need to be careful with the dimensions.
- PyTorch is a blessing because it lets us ignore all of this complexity by automatically taking care of the derivatives. We only need to define what a forward pass is and PyTorch knows how to differentiate with respect to every single parameter in the network.
- The proper way to do this would be with Einstein notation (*link*), which uses implicit sums over repeated indices (but let's stop here).

Simple Network

Of course, none of this matrix nonsense matters if we are only interested in the value of the gradient for one single parameter in the network (**z^1 with respect to W^1_{21}, for example**). In this case we know how the parameter is calculated and we can safely ignore any complicated tensor operations, just by explicitly expanding the dependencies of the particular parameter with respect to the others.

z^1 with respect to W^1_{21} example:

$$z^1 = W^{1\top} x = \begin{pmatrix} W^1_{11} & W^1_{12} \\ W^1_{21} & W^1_{22} \\ W^1_{31} & W^1_{32} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} z^1_1 \\ z^1_2 \\ z^1_3 \end{pmatrix} \longrightarrow \frac{\partial z^1}{\partial W^1_{21}} = \frac{\partial (W^1_{21} x_1 + W^1_{22} x_2)}{\partial W^1_2 1} = x_1$$



3-Regularisation, Bias, and Variance

Lluis Guasch

GTAs online:
Raul
Alex

Objectives of the day:

- Underfitting and overfitting in relation to bias and variance
- Regularization to control variance/overfitting
- Batch normalization
- Machine learning diagnostics for hyperparameter tunning
- Training, validation, and test sets
- k-fold validation
- Hyperparameter search strategies

Regularisation, Bias, and Variance

1. Overfitting and underfitting, bias and variance
2. Regularization
3. Batch normalization
4. Machine learning diagnostics
5. Training, validation, and test sets
6. K-fold validation

1. **Overfitting and underfitting, bias and variance**
2. Regularization
3. Batch normalization
4. Machine learning diagnostics
5. Training, validation, and test sets
6. K-fold validation

OVERFITTING

The training set error is **not** a good indicator of network performance, why?

OVERFITTING

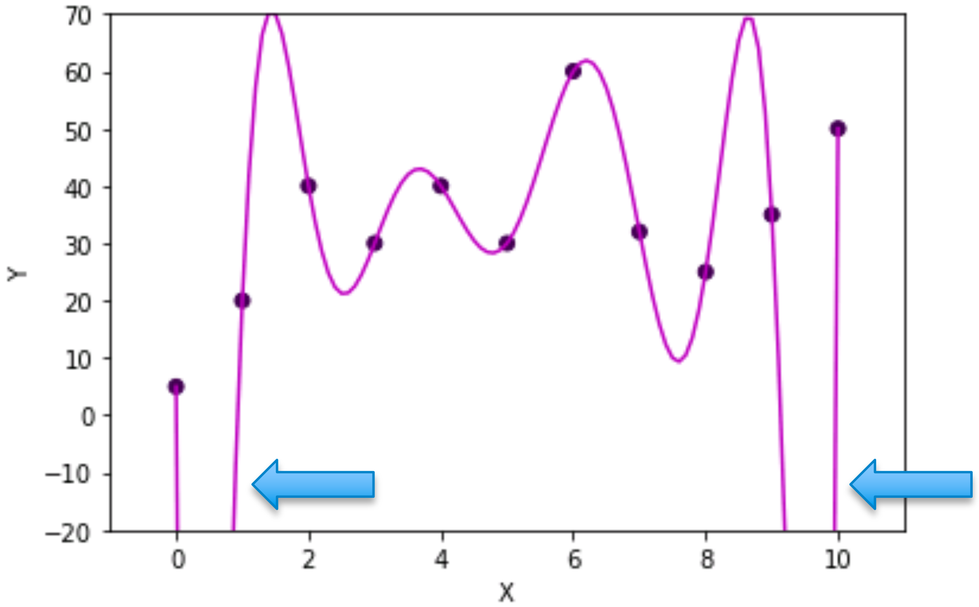
The training set error is **not** a good indicator of network performance, why?

If we have a large number of fitting parameters, the trained model may fit the training set very well:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 = 0$$

But fail to generalize to new data! This is called *Overfitting.*

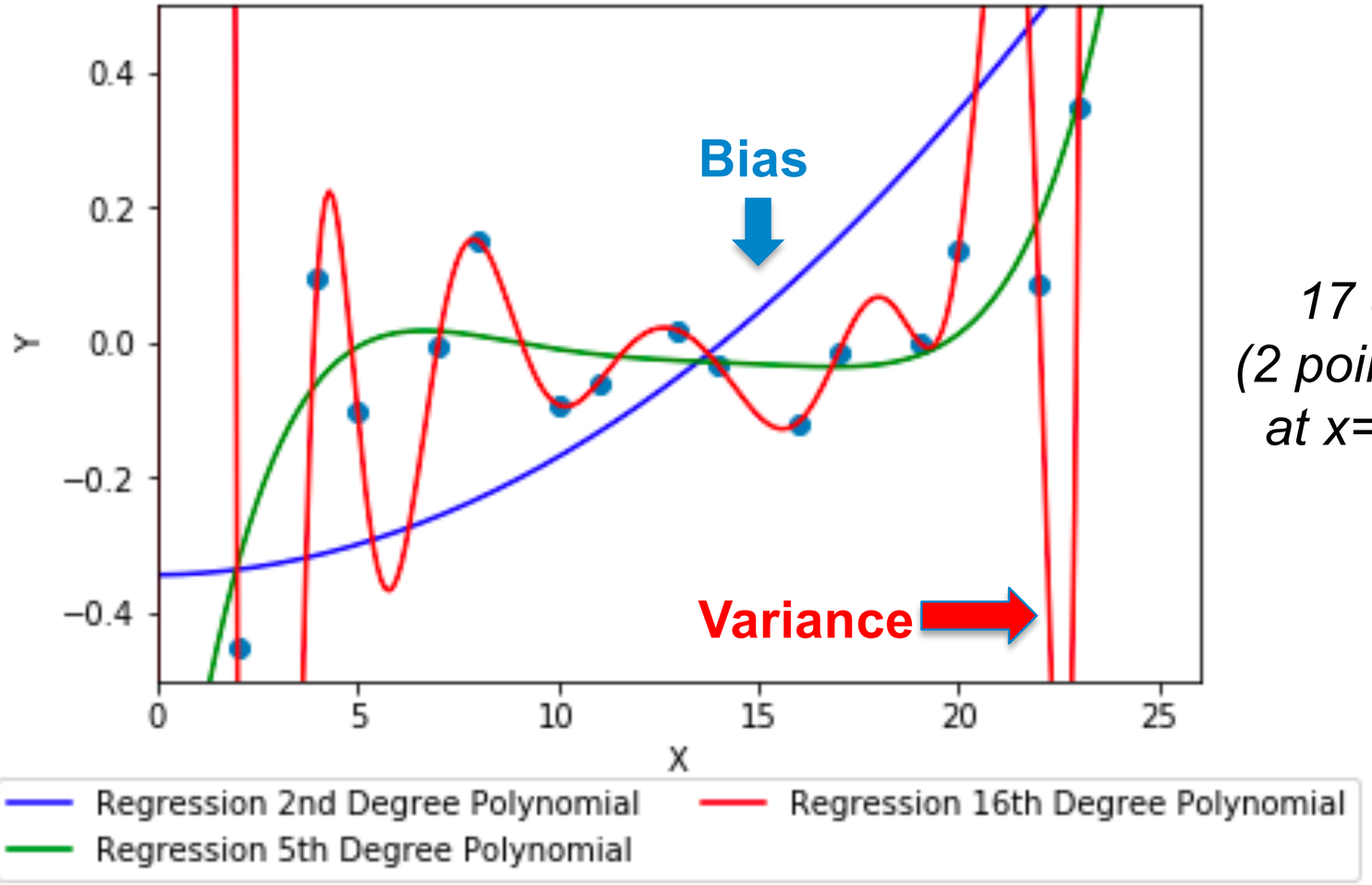
Example of Polynomial Overfitting



$m = 11$ data points !

10th degree polynomial with bias term

Bias (or underfitting) VS variance (or overfitting)



17 data points
(2 points not shown
at x=0 and x=26)

Bias and **variance** etymology:

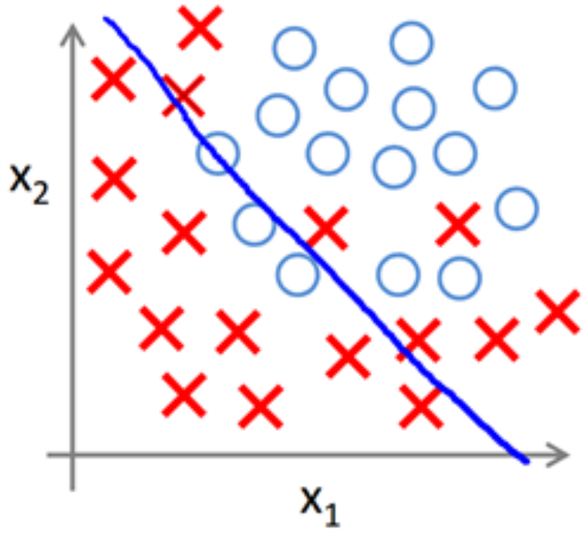
Bias: *the hypothesis function $h_\theta(x)$ has a « pre-conception » or an « a priori » idea of the data variations which is too simple, which is biased from the start, considering the actual variability of the data (for instance it is a polynomial of too low degree).*

Variance: *the hypothesis function $h_\theta(x)$ has too many degrees of freedom – or parameters - and, as a result, can fit too many possible functions, with too much variance, considering the actual variability of the data. (for instance it is a polynomial of too high degree)*

Bias and Variance

We have already seen examples of overfitting and underfitting:

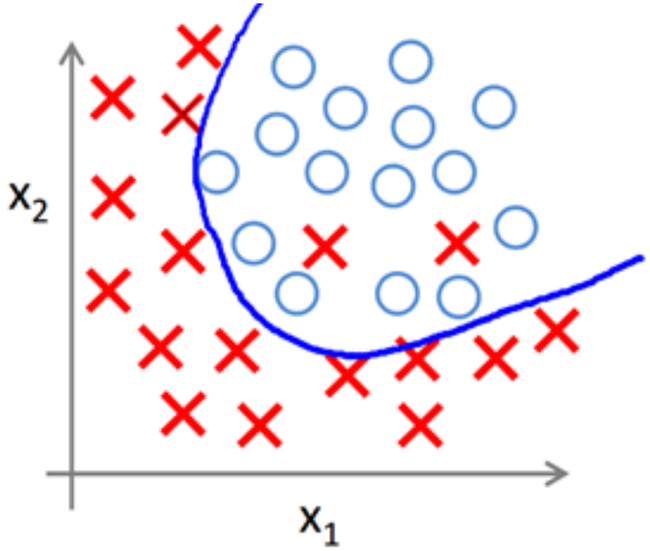
Bias



$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$

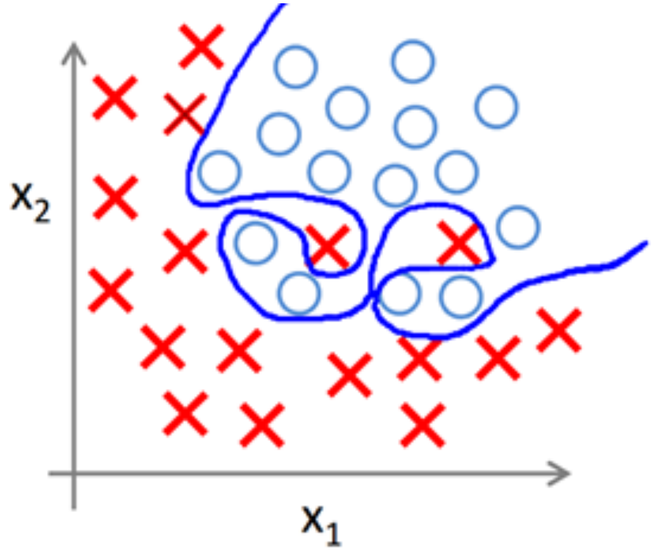
(g = sigmoid function)

OK



$g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2 + \theta_5 x_1 x_2)$

Variance

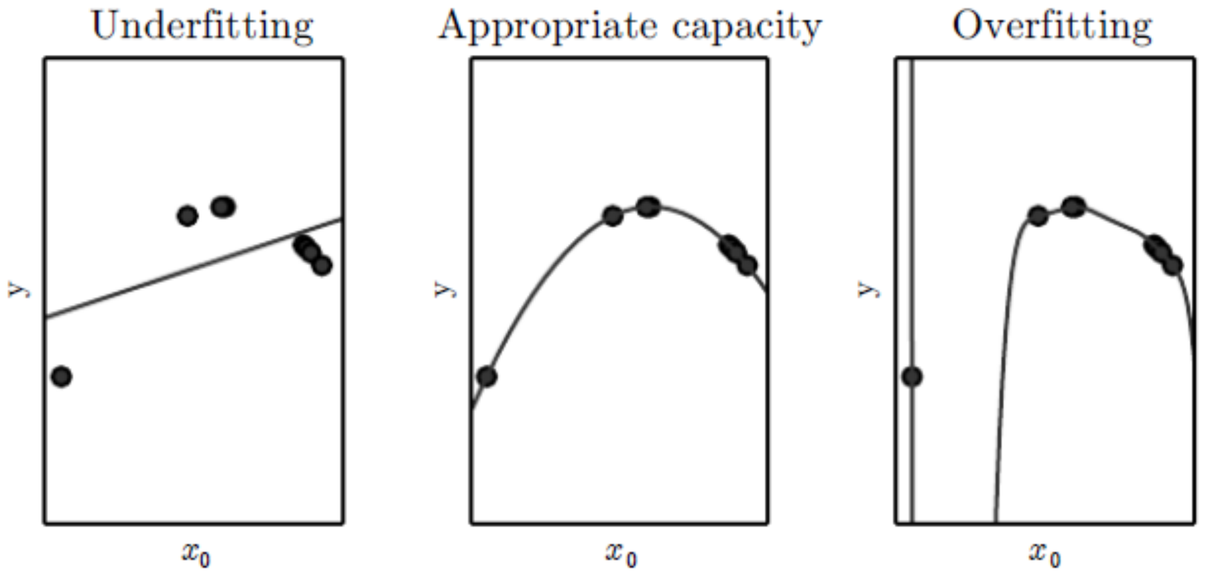


$g(\theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^2 x_2 + \theta_4 x_1^2 x_2^2 + \theta_5 x_1^2 x_2^3 + \theta_6 x_1^3 x_2 + \dots)$

Source: Machine Learning Course, Andrew Ng

We define the **capacity** of the model as:

*The **Capacity** of a model is associated with its likely underfitting or overfitting. Informally a model's capacity is its ability to fit a wide variety of functions. Models with low capacity may tend to **underfit (Bias) because they do not have enough parameters** , models with high capacity may tend to **overfit (Variance) because they have too many parameters.***



Source: Goodfellow et al, 2017

Generalisation definition:

*The trained model must perform well on new, previously unseen data, not just those on which the model was trained. The ability to perform well on **previously unseen data** – or **Test Data** - is called **Generalization**.*

Source: Goodfellow et al, 2017

Bias and Variance

A good machine learning algorithm should:

1. Make the training error small. If this is not the case, we have **underfitting** or **bias**.

2. Make the error gap between the training and test set small. If this is not the case, we have **overfitting** or **variance**.

Source: Goodfellow et al, 2017

Regularisation, Bias, and Variance

1. Overfitting and underfitting, bias and variance
2. **Regularization**
3. Batch normalization
4. Machine learning diagnostics
5. Training, validation, and test sets
6. K-fold validation

How to avoid overfitting? With **regularization**:

Regularization is any modification we make to a learning algorithm that is intended to reduce its Generalization Error but not its Training Error...

*An effective regularizer is one that **reduces Variance** significantly while **not increasing the Bias**.*

Source: Goodfellow et al, 2017

L1 and L2 regularizers:

Loss

L1 and L2 Regularizations consist of adding a new term to the objective function in order to control the variations of the parameters:

$$J(\theta) = \frac{1}{2m}\left(\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 + \lambda \sum_{j=1}^{n} \theta_j^2\right)$$ if L2 norm is used

Regularization Parameter, controlling the "Weight Decay"

$$J(\theta) = \frac{1}{2m}\left(\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 + \lambda \sum_{j=1}^{n} |\theta_i|\right)$$ if L1 norm is used

Ridge and Lasso regressions:

$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$$

Ridge Regression **(preferred because math derivations simpler)**

$$J(\theta) = \frac{1}{2m}\left(\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 + \lambda \sum_{j=1}^{n} \theta_j^2\right)$$ if L2 norm is used

Tikhonov regularisation

Lasso Regression

$$J(\theta) = \frac{1}{2m}\left(\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 + \lambda \sum_{j=1}^{n} |\theta_i|\right)$$ if L1 norm is used

Exercise: classification with L2 regularisation

Take the Case of Binary Linear Logistic Regression in 2-D.

Write $h_\theta(x)$ *as a function of* x *and* θ.

Assume there are m data points $\left(x^{(i)}, y^{(i)}\right)$ *for* $i = 1, \dots, m$

Express $J(\theta)$*using L2 regularization and a regularization parameter* λ.

Calculate $\frac{\partial J(\theta)}{\partial \theta_j}$ *for one of the* θ_j *parameters*

Exercise: classification with L2 regularisation

Take the Case of Binary Linear Logistic Regression in 2-D.

Write $h_\theta(x)$ *as a function of* x *and* θ. $h_\theta(x) = \sigma(\theta_0 + \theta_1 x_1 + \theta_2 x_2) = \frac{1}{1 + e^{-(\theta_0 + \theta_1 x_1 + \theta_2 x_2)}}$

Assume there are m data points $\left(x^{(i)}, y^{(i)}\right)$ *for* $i = 1, \ldots, m$

Express $J(\theta)$ *using L2 regularization and a regularization parameter* λ.

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(h_\theta\left(x^{(i)}\right)\right) + \left(1 - y^{(i)}\right)\log\left(1 - h_\theta\left(x^{(i)}\right)\right)\right] + \frac{\lambda}{2m}(\theta_0^2 + \theta_1^2 + \theta_2^2)$$

Calculate $\frac{\partial J(\theta)}{\partial \theta_j}$ *for one of the* θ_j *parameters:* $\frac{\partial J(\theta)}{\partial \theta_2} = \frac{1}{m}\sum_{i=1}^{m}\left[h_\theta\left(x^{(i)}\right) - y^{(i)}\right]x_2^{(i)} + \frac{\lambda}{m}\theta_2$

Gradient descent for regularised logistic regression:

Gradient-Descent Approach without Regularization Term

$$\theta_j := \theta_j - \alpha \frac{\partial J}{\partial \theta_j} := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right) x_j^{(i)}$$

Gradient-Descent with Regularization Term (if L2 norm is used)

$$\theta_j := \theta_j - \alpha \frac{\partial J}{\partial \theta_j} := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right) x_j^{(i)} - \alpha \frac{\lambda}{m} \theta_j$$

$$\theta_j := \theta_j \left(1 - \alpha \frac{\lambda}{m} \right) - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right) x_j^{(i)}$$

Gradient descent for regularised logistic regression:

Gradient-Descent Approach without Regularization Term

$$\theta_j := \theta_j - \alpha \frac{\partial J}{\partial \theta_j} := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)} \right) x_j^{(i)}$$

Gradient-Descent with Regularization Term (if L2 norm is used)

$$\theta_j := \theta_j - \alpha \frac{\partial J}{\partial \theta_j} := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)} \right) x_j^{(i)} - \alpha \frac{\lambda}{m} \theta_j$$

$$\theta_j := \theta_j \left(1 - \alpha \frac{\lambda}{m}\right) - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)} \right) x_j^{(i)}$$

Gradient descent for regularised logistic regression:

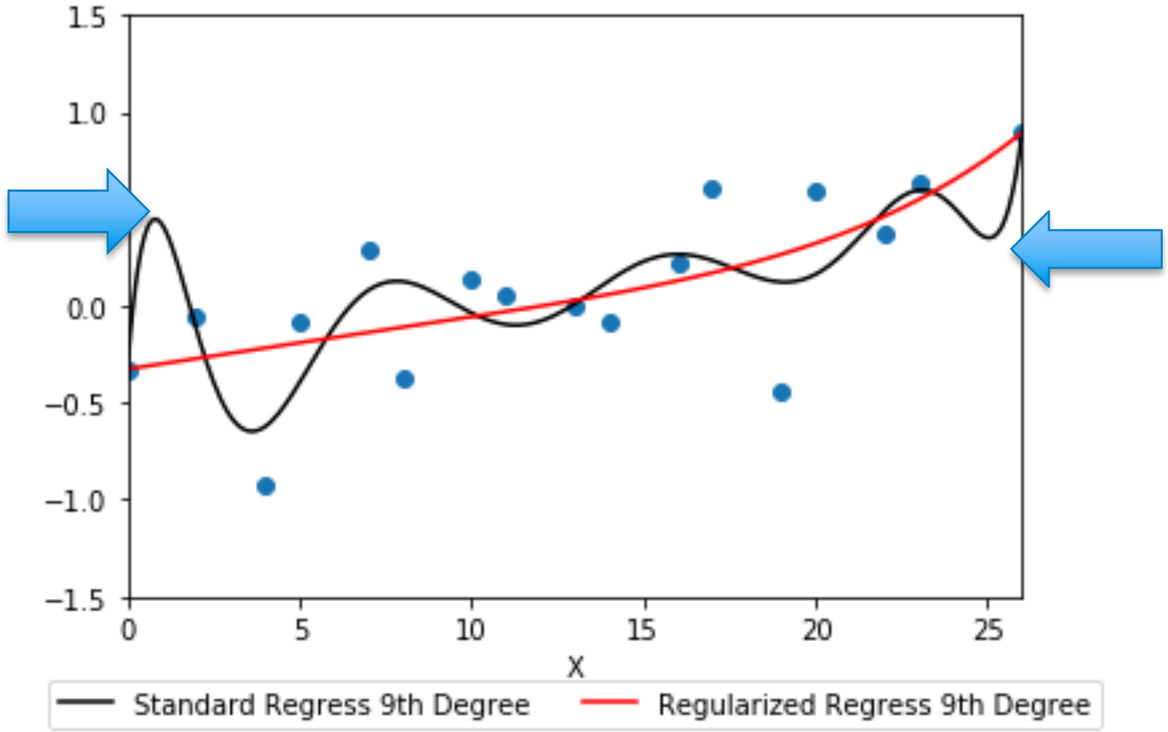
Consider in more detail the Gradient Descent term in case of Regularization:

$$\theta_j := \theta_j \left(1 - \alpha \frac{\lambda}{m}\right) - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right) x_j^{(i)}$$

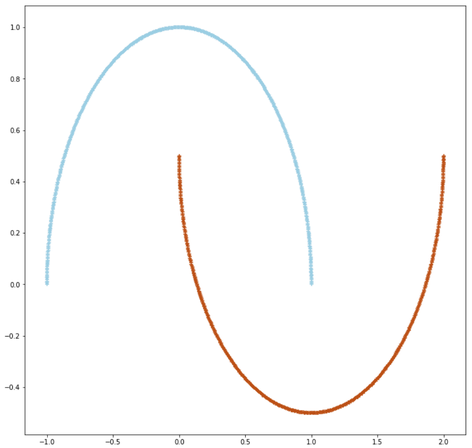
Systematic decrease of absolute value θ_j at each iteration

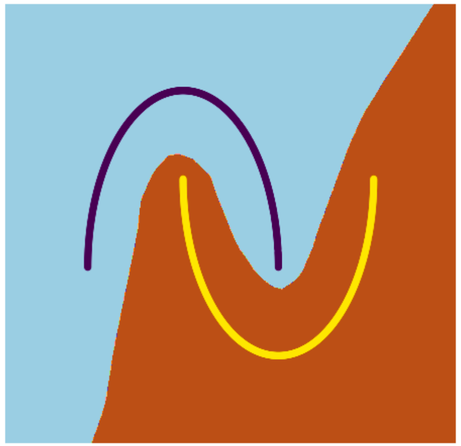
Same term as for optimization without regularization

Effect of regularisation in regression problems:



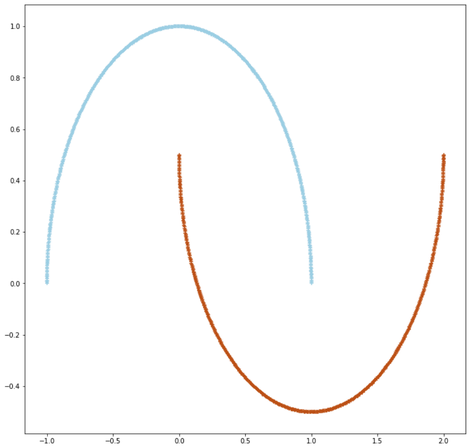
Regularisation effects in neural networks:

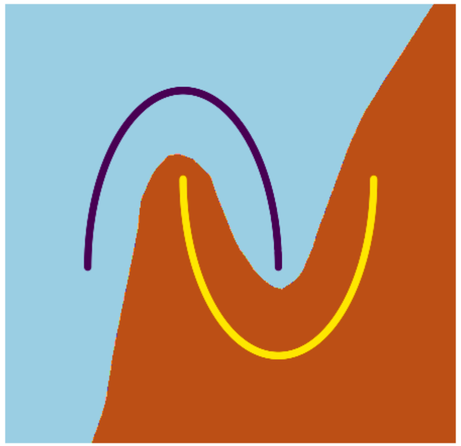




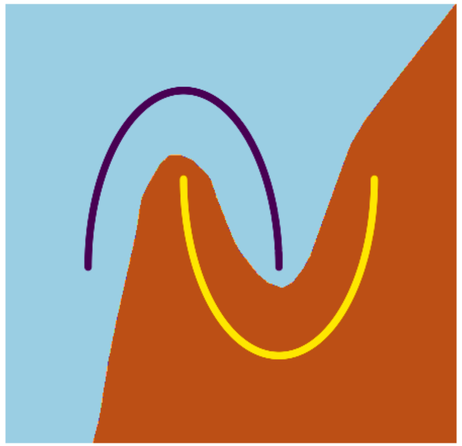
$\lambda = 0$

Regularisation effects in neural networks:

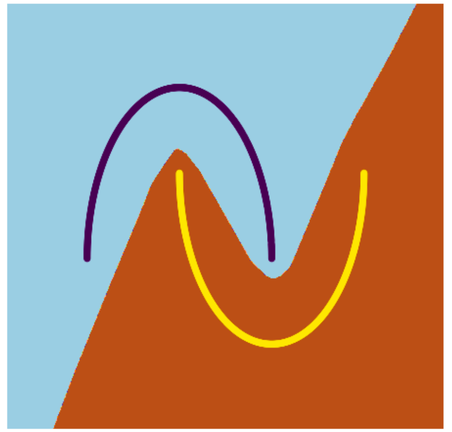




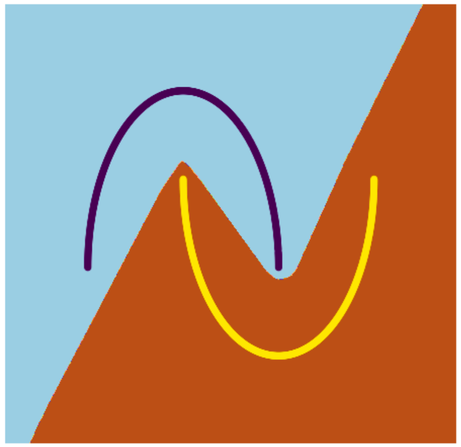
$\lambda = 0$



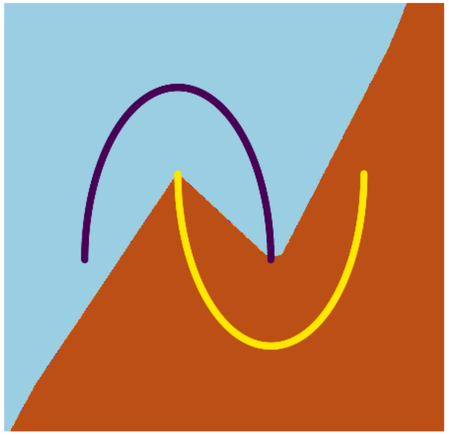
$\lambda = 0.1$



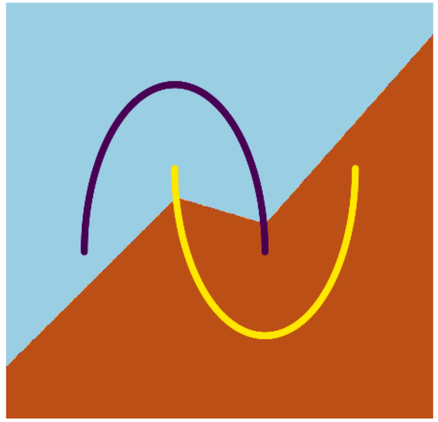
$\lambda = 1$



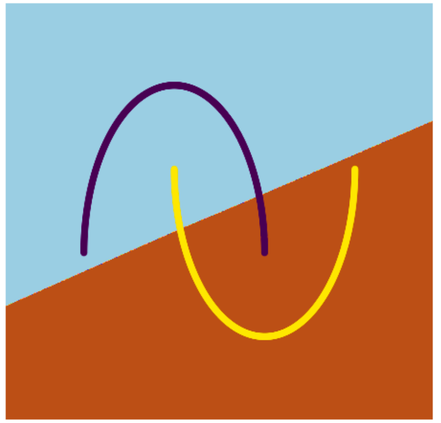
$\lambda = 2.5$



$\lambda = 5$



$\lambda = 7.5$

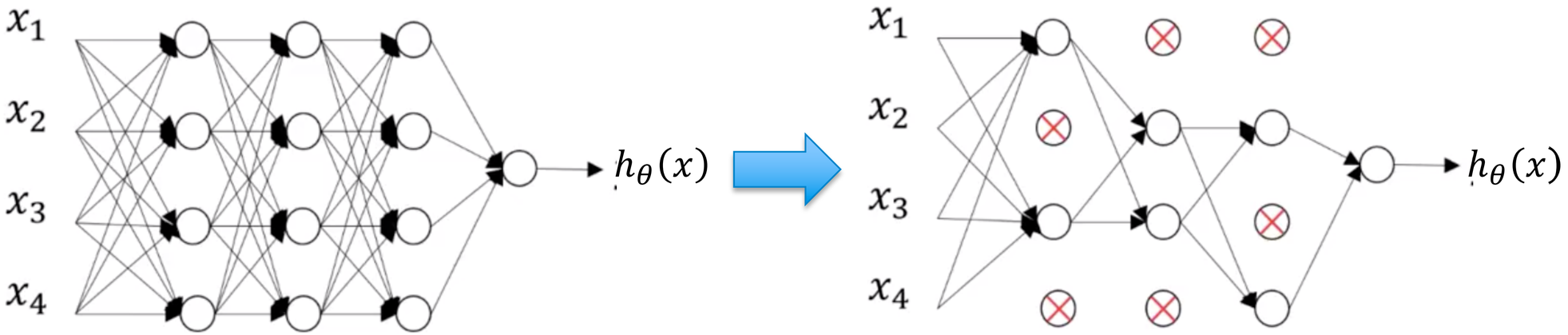


$\lambda = 10$

DROPOUT REGULARISATION

There are other strategies to regularise our networks. **Dropout**:

At Training time:



For each Training example… Drop hidden layers units with 0.5 (or p) probability

Same approach for input layer but with p closer to 1. No change in output layer.

We are averaging the results over a set of network configurations!

Dropout regularisation

A short video by Hinton explaining dropout:

https://www.youtube.com/watch?v=kAwF--GJ-ek

Original **dropout** paper has more than 27k citations (that's quite a lot):

Journal of Machine Learning Research 15 (2014) 1929-1958 Submitted 11/13; Published 6/14

Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava NITISH@CS.TORONTO.EDU
Geoffrey Hinton HINTON@CS.TORONTO.EDU
Alex Krizhevsky KRIZ@CS.TORONTO.EDU
Ilya Sutskever ILYA@CS.TORONTO.EDU
Ruslan Salakhutdinov RSALAKHU@CS.TORONTO.EDU
Department of Computer Science
University of Toronto
10 Kings College Road, Rm 3302
Toronto, Ontario, M5S 3G4, Canada.

Editor: Yoshua Bengio

https://jmlr.csail.mit.edu/papers/volume15/srivastava14a/srivastava14a.pdf

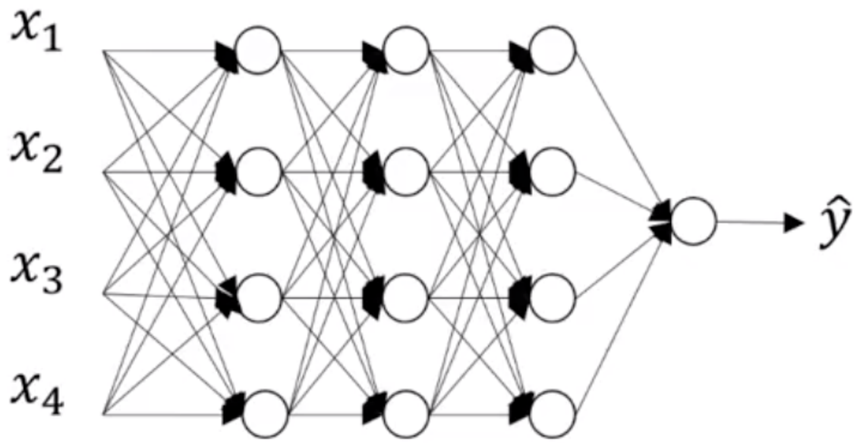
DROPOUT REGULARISATION

What happens after we have finished training with **dropout**?

DROPOUT REGULARISATION

What happens after we have finished training with **dropout**?

At Test time: no drop-out, use all hidden units but multiply their output by $p = 0.5$



Drop-out can also be used to quantify Model Uncertainty in Bayesian Neural Networks (Gal and Ghahramani, 2016)

Dropout can be interpreted as a form of **ensemble learning**:

Given a Training Set of m data, train not just one but a number k of different models and predict the Test Set by averaging the results of the k models.

The k models can consist of the same algorithm trained on different subsets of the initial Training Set (this is called Bagging), or different algorithms trained on the same Training Set (this is Model Averaging).

Data augmentation can also be used as a regularisation technique:

Data augmentation can also be used as a regularisation technique:

“Regularization adds prior knowledge to a model […]. Augmentation is also a form of adding prior knowledge to a model; e.g. images are rotated, which you know does not change the class label.”

https://stats.stackexchange.com/questions/295383/why-is-data-augmentation-classified-as-a-type-of-regularization

Data Augmentation

Data augmentation **strategies**:

- Rotation (to some reasonable degree)
- Translation (up, down, left, right)
- Zooming
- Cropping
- Added noise
- Changing the Brightness level

…

These are readily available in Python code (*imgaug, Albumentation* packages…).

1. Overfitting and underfitting, bias and variance
2. Regularization
3. **Batch normalization**
4. Machine learning diagnostics
5. Training, validation, and test sets
6. K-fold validation

Batch Normalisation

There are different interpretations of what **batch normalisation** does.

Some people interpret it as a means to reduce the **internal covariate shift**, but more recent developments seem to point that it acts as a **regulariser** that smooths the objective function (and therefore make it better suited to gradient descent minimisation methods).

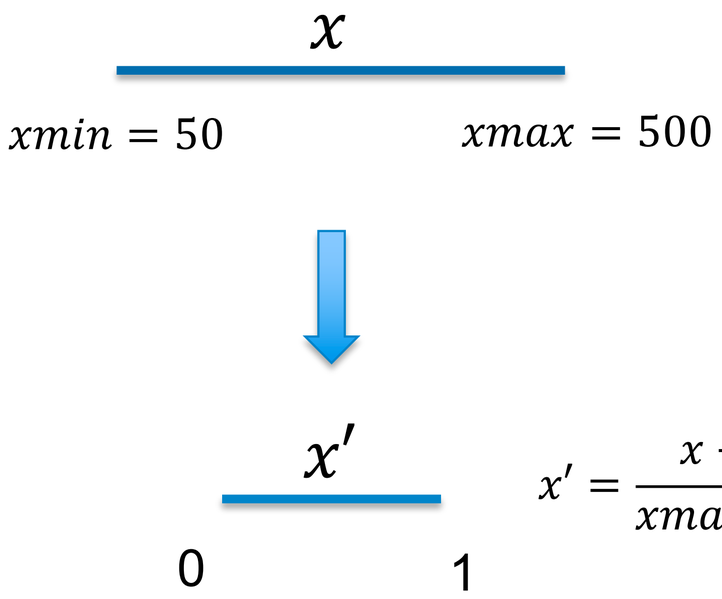
- **Internal covariate shift**: Each layer of a neural network has inputs with a corresponding distribution, which is affected during the training process by the randomness in the parameter initialization and the randomness in the input data. The effect of these sources of randomness on the distribution of the inputs to internal layers during training is described as internal covariate shift.

https://en.wikipedia.org/wiki/Batch_normalization#Internal_covariate_shift

Definitions:

- all data values between 0 and 1

Normalization



$$x' = \frac{x - xmin}{xmax - xmin}$$

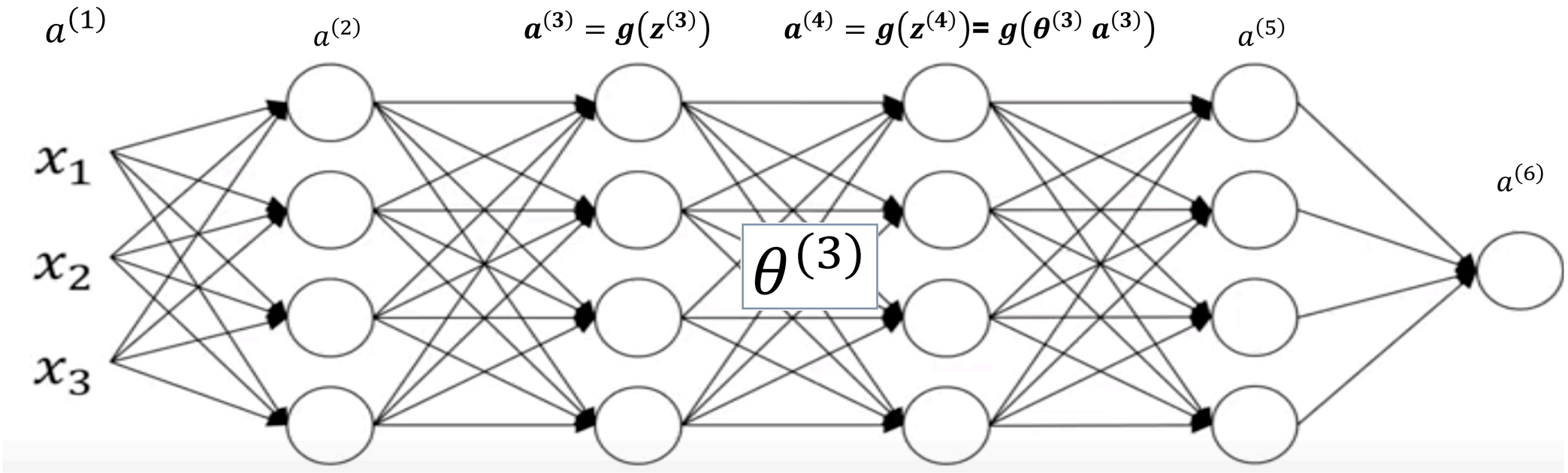
- mean is 0
- standard deviation is 1

Standardization

$$x' = \frac{x - \mu_x}{\sigma_x}$$

(where μ_x and σ_x are mean and standard deviation of array x)

Batch normalisation is applied to hidden layers:



Batch normalisation on data at each layer of the network ($z^{(l)}$):

1. Normalize $z^{(l)}$ vector at layer (l):

$$z_{norm}^{(l)} = \frac{z^{(l)} - \mu_z}{\sqrt{\sigma_z^2 + \varepsilon^2}}$$

(with μ_z and σ_z^2 mean and variance of $z^{(l)}$ calculated separately on each mini-batch, and ε small parameter useful if σ_z^2 = 0)

2. Apply Linear Transform to $z_{norm}^{(l)}$, with trainable parameters $\beta^{(l)}$ and $\gamma^{(l)}$. For each component j of $z_{norm}^{(l)}$

$$\tilde{z}_j^{(l)} = \gamma_j^{(l)}\, z_{j\,norm}^{(l)} + \beta_j^{(l)}$$

→ Optimize mean and variance of $\tilde{z}^{(l)}$

3. Use $\tilde{z}^{(l)}$ as input to the activation function: $a^{(l)} = g\left(\tilde{z}^{(l)}\right)$

Batch normalisation on data at each layer of the network ($z^{(l)}$):

1. Normalize $z^{(l)}$ vector at layer (l):

   Standardize

$$z_{norm}^{(l)} = \frac{z^{(l)} - \mu_z}{\sqrt{\sigma_z^2 + \varepsilon^2}}$$

(with μ_z and σ_z^2 mean and variance of $z^{(l)}$ calculated separately on each mini-batch, and ε small parameter useful if σ_z^2 = 0)

2. Apply Linear Transform to $z_{norm}^{(l)}$, with trainable parameters $\beta^{(l)}$ and $\gamma^{(l)}$. For each component j of $z_{norm}^{(l)}$

$$\tilde{z}_j^{(l)} = \gamma_j^{(l)}\ z_{j\,norm}^{(l)} + \beta_j^{(l)}$$

→ Optimize mean and variance of $\tilde{z}^{(l)}$

3. Use $\tilde{z}^{(l)}$ as input to the activation function: $a^{(l)} = g\left(\tilde{z}^{(l)}\right)$

BATCH NORMALISATION

Batch normalisation original paper (more than 21k citations):

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe
Google Inc., *sioffe@google.com*

Christian Szegedy
Google Inc., *szegedy@google.com*

Abstract

Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities. We refer to this phenomenon as *internal covariate shift*, and address the problem by normalizing layer inputs. Our method draws its strength from making normalization a part of the model architecture and performing the normalization *for each training mini-batch*. Batch Normalization allows us to use much higher learning rates and be less careful about initialization. It also acts as a regularizer, in some cases eliminating the need for Dropout. Applied to a state-of-the-art image classification model, Batch Normalization achieves the same accuracy with 14 times fewer training steps, and beats the original model by a significant margin. Using an ensemble of batch-normalized networks, we improve upon the best published result on ImageNet classification: reaching 4.9% top-5 validation error (and 4.8% test error), exceeding the accuracy of human raters.

Using mini-batches of examples, as opposed to one example at a time, is helpful in several ways. First, the gradient of the loss over a mini-batch is an estimate of the gradient over the training set, whose quality improves as the batch size increases. Second, computation over a batch can be much more efficient than m computations for individual examples, due to the parallelism afforded by the modern computing platforms.

While stochastic gradient is simple and effective, it requires careful tuning of the model hyper-parameters, specifically the learning rate used in optimization, as well as the initial values for the model parameters. The training is complicated by the fact that the inputs to each layer are affected by the parameters of all preceding layers – so that small changes to the network parameters amplify as the network becomes deeper.

The change in the distributions of layers' inputs presents a problem because the layers need to continuously adapt to the new distribution. When the input distribution to a learning system changes, it is said to experience *covariate shift* (Shimodaira, 2000). This is typically handled via domain adaptation (Jiang, 2008). However, the notion of covariate shift can be extended beyond the learning system as a whole, to apply to its parts, such as a

and a good video by Andrew Ng:

https://www.youtube.com/watch?v=tNIpEZLv_eg

Batch normalisation summary:

- Performs pre-processing at each layer of the network, instead of just on input layer.
- Accelerates training by stabilizing the layer changes through the iterations.
- Provides some regularization, like dropout, but not as strong (open topic of discussion).
- Batch normalisation should really be called batch standardization.
- It reduces the probability that vanishing or exploding gradients occur.
- Variations on this concept include **layer normalization** (developed by Hinton et al).

Regularisation, Bias, and Variance

1. Overfitting and underfitting, bias and variance
2. Regularization
3. Batch normalization
4. **Machine learning diagnostics**
5. Training, validation, and test sets
6. K-fold validation

In the previous sections we have learned about Bias and Variance problems in our networks, and how to fix them.

But how do we identify them in the first place?

Accuracy

In classification problems we use cross-entropy as a loss, but to assess the quality of the resulting models we will use a new metric: **Accuracy**

$$Accuracy = \frac{Number\ of\ Correctly\ Classified\ Examples}{Total\ Number\ of\ Examples}$$

Why do we use cross-entropy instead of accuracy?

Accuracy

In classification problems we use cross-entropy as a loss, but to assess the quality of the resulting models we will use a new metric: **Accuracy**

$$Accuracy = \frac{Number\ of\ Correctly\ Classified\ Examples}{Total\ Number\ of\ Examples}$$

Why do we use cross-entropy instead of accuracy?

Because it is easily **differentiable** and can be used in gradient-descent-based training.

CONFUSION MATRIX

We can also analyse the distribution of misclassifications:

Predict	0	1	2	3	4	5	6	7	8	9
Actual										
0	585	1	1	0	0	1	1	0	2	1
1	0	666	2	2	0	0	0	0	2	2
2	1	10	569	2	1	0	3	6	3	1
3	3	2	6	578	0	13	0	2	6	3
4	1	2	7	0	561	1	1	1	2	8
5	1	0	0	7	2	521	3	0	6	2
6	8	0	1	0	8	2	569	1	3	0
7	2	5	6	3	3	2	0	595	1	10
8	3	8	5	7	2	9	5	0	539	7
9	5	2	0	4	9	3	0	8	7	557

Example of **confusion matrix** in MNIST

CONFUSION MATRIX

The confusion matrix for a binary problem is simply:

		Predicted	
		Negative	**Positive**
Actual	**Negative**	True Negative	False Positive
	Positive	False Negative	True Positive

		Predicted/Classified	
		Negative	**Positive**
Actual	**Negative**	998	0
	Positive	1	1

What is the accuracy of this model? $\frac{998+1}{1000} = 0.999$

What if the false negative corresponds to a person sick with COVID, to a major fraud case or to a dangerous terrorist? ***The impact may be big!!***

Precision and Recall

The confusion matrix for a binary problem is simply:

$$\text{Precision} = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

		Predicted: Negative	Predicted: Positive
Actual	Negative	True Negative	False Positive
	Positive	False Negative	True Positive

Precision is the proportion of predicted positive which are properly predicted. Good if you want to minimize the number of false positives (example of spam detection).

$$\text{Recall} = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

		Predicted: Negative	Predicted: Positive
Actual	Negative	True Negative	False Positive
	Positive	False Negative	True Positive

Recall is the proportion of true positive which are properly predicted. Good if you want to minimize the number of false negative (example of contagious patient or terrorist detection).

https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9

F1 Score

The F1 score is the harmonic mean of Precission and Recall:

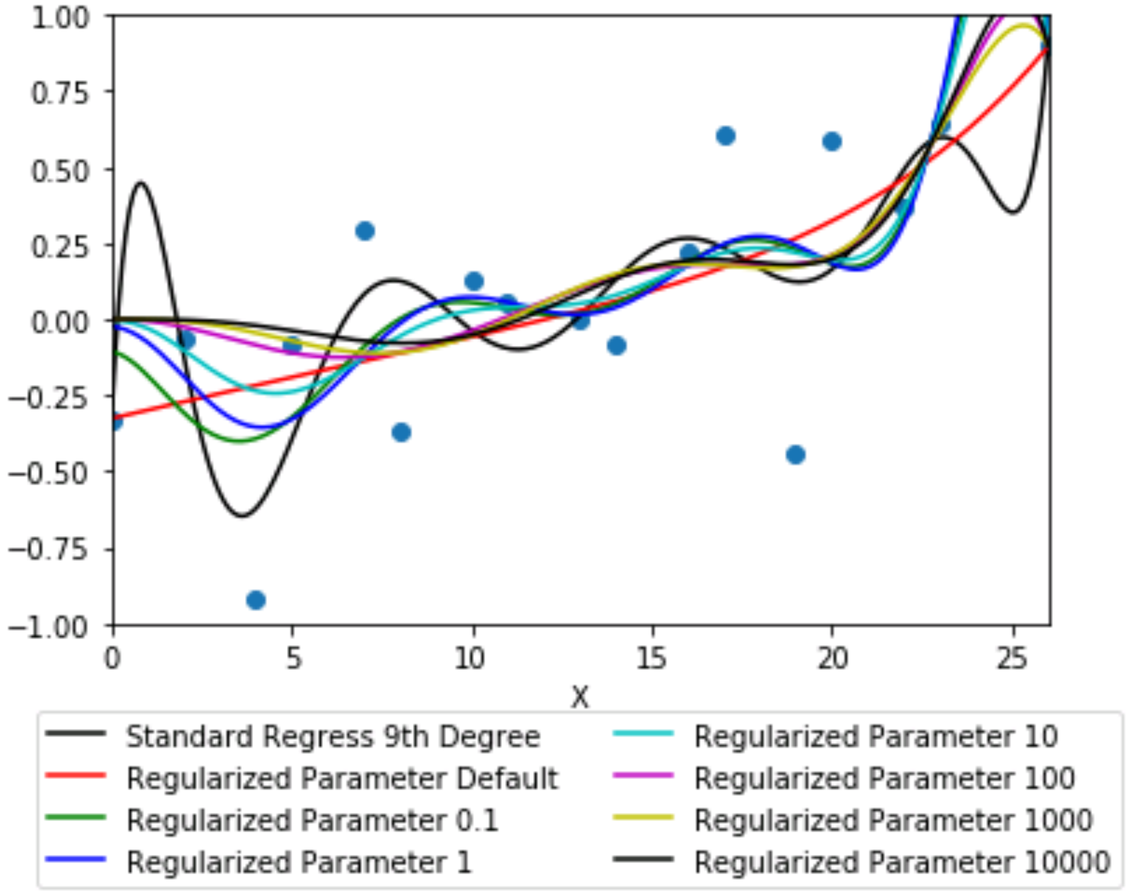
$$\frac{1}{F1} = \frac{1}{2}\left(\frac{1}{Precision} + \frac{1}{Recall}\right)$$

The highest possible value of F_1 is 1, indicating perfect precision and recall, and the lowest possible value is 0, if either the precision or the recall is zero.

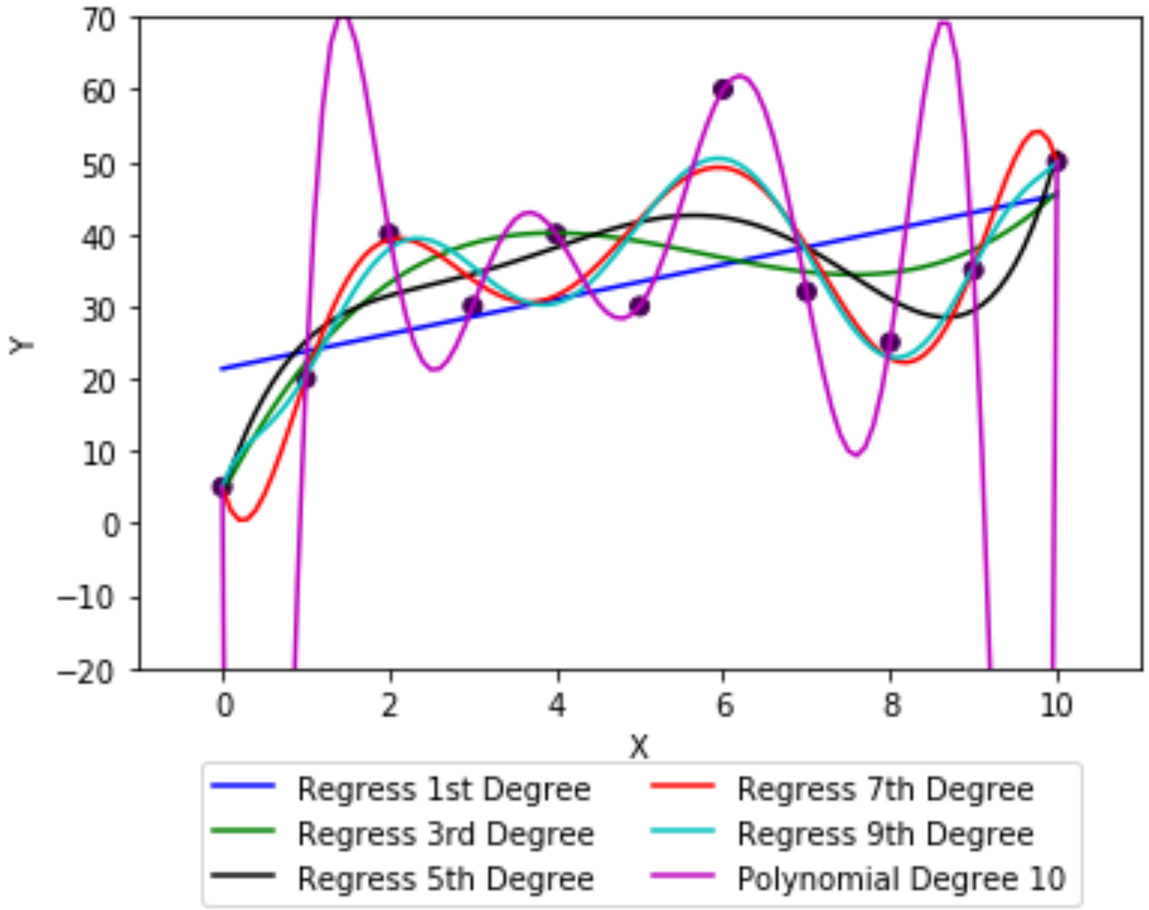
OTHER DIAGNOSTIC TOOLS

Besides measuring accuracy, what else can we do to understand the performance of our networks?

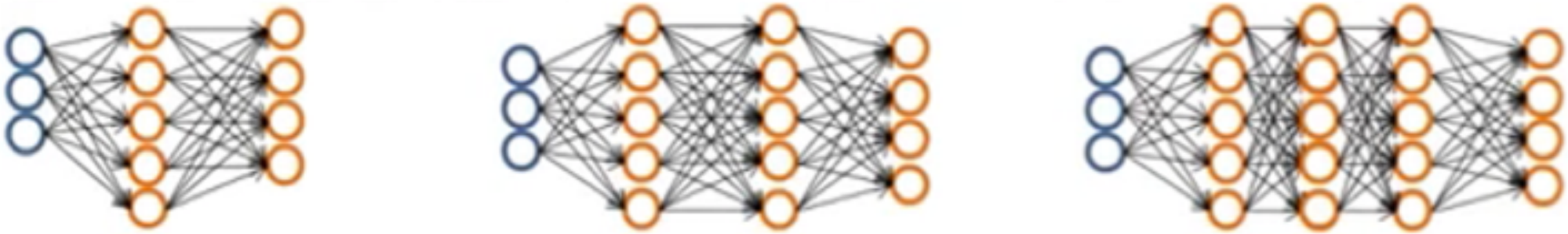
How can we choose the **best regularisation parameters**?:



How can we choose the **best polynomial degree**?:



How can we choose the **best network architecture**?:



Machine learning diagnostic (informal definition):

A test to gain insight about the performance of a training algorithm

Machine learning diagnostic (informal definition):

A test to gain insight about the performance of a training algorithm

We will use the data by splitting it into different sets

Regularisation, Bias, and Variance

1. Overfitting and underfitting, bias and variance
2. Regularization
3. Batch normalization
4. Machine learning diagnostics
5. **Training, validation, and test sets**
6. K-fold validation

What is a hyperparameter?

A **hyperparameter** is a neural network parameter whose value is set before the Training process begins. It does not change during Training. By contrast, the values of parameters θ are derived via Training.

Exercise: Give Examples of Hyper-Parameters of a Neural Network

What is a hyperparameter?

A **hyperparameter** is a neural network parameter whose value is set before the Training process begins. It does not change during Training. By contrast, the values of parameters θ are derived via Training.

Exercise: Give Examples of Hyper-Parameters of a Neural Network

Batch Size, Learning Rate, Optimization Approach used, Number of Layers, Number of Hidden Units, Regularization Parameter…

But how can we pick the optimal hyperparameters?

Validation Set

The need for a **validation set**:

In a Supervised Learning context, the Neural Network parameters (or weights) are trained on the Training Set, but tested on the Test Set.

The Test Set constitutes the unseen data, it should not be used to calculate the parameters of the Neural Network … or to optimize the Hyperparameters.

Hence, to optimize the Hyperparameters, we need a third Set, the Validation Set

Validation Set

Create a **validation set** to optimise hyperparameters:

Split Data Into Three Sets:
Training Set $(\sim 70\%)$, *Validation Set* $(\sim 15\%)$, *Test Set* $(\sim 15\%)$

Example of MNIST: *the Training Set contains 60,000 images, the « official » Test Set contains 10,000 images. This ratio is usually appropriate for this size of datasets. For very large datasets (say 100.000's to millions), split between Training and Test Set sizes can be smaller and be as low as 90%-10%.*
Note that MNIST does not have a pre-defined Validation Set. This has to be chosen and extracted from the Training Set by the user.

Validation Set

Use a **validation set** to optimise hyperparameters:

For each Possible Choice of Neural Network Hyperparameters:

1. Train Neural Network Parameters on Training Set
2. Test Performance on the Validation Set
3. Pick the Hyperparameters that give the best performance on the Validation Set.
4. Possibly retrain the new Neural Network on Training+Validation Set.
5. Test the Neural Network on the Test Set

The Test Set is the final measure of performance but must never be used in the Training!

Test Set

Warning: what would happen if we use the test set during training?

Test Set

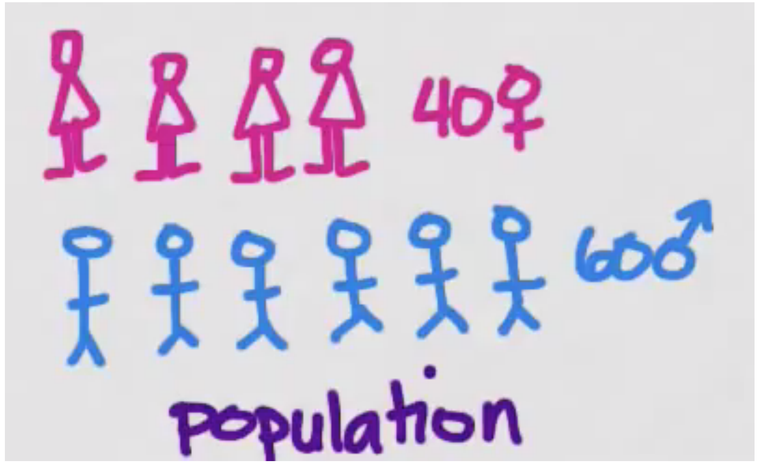
Warning: what would happen if we use the test set during training?

The **test set error** will **NOT** be representative of the **generalisation error**!

Sampling the validation set (and possibly the test set) to avoid class imbalance:

In Classification problems, make sure the proportions of each class are the same for Training, Validation and possibly Test Sets. Take the example of a 20% Validation Set.

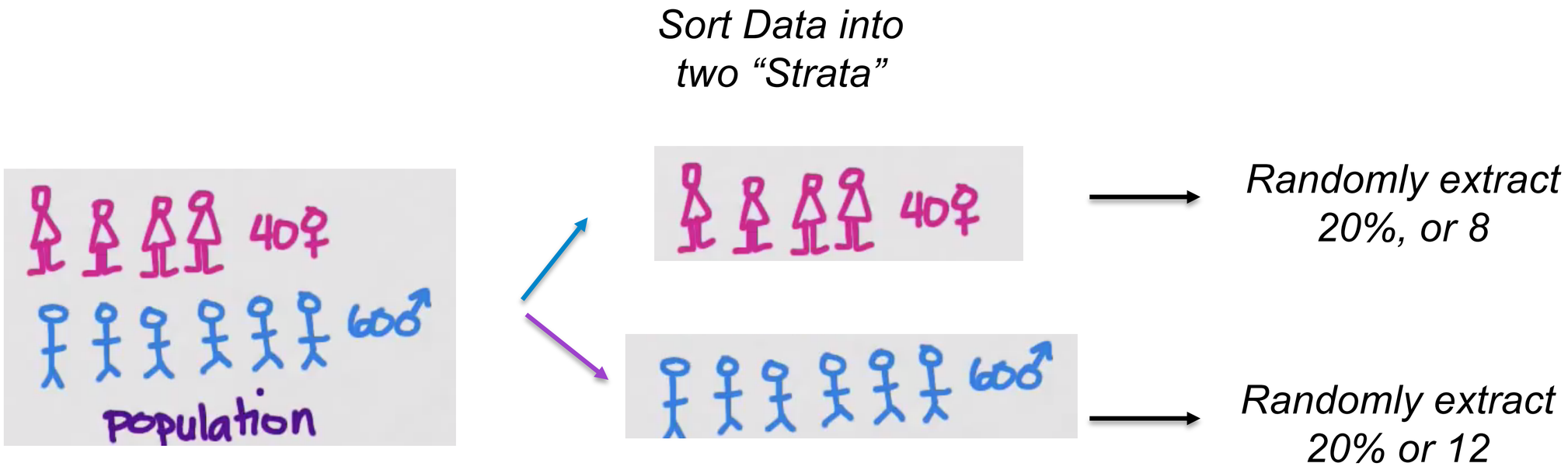
How to create a 20% Validation Set from this population?



If I randomly select 20 individuals (that is 20%) from this group of 40 women and 60 men, it is very unlikely that the group of 20 will be exactly composed of 8 (that is 20%) women and 12 (that is 20%) men!

For smaller data sets, use Stratified Random Sampling (class by class) instead of global sampling over the whole set.

Stratified sampling: create a dataset with 20% of the data

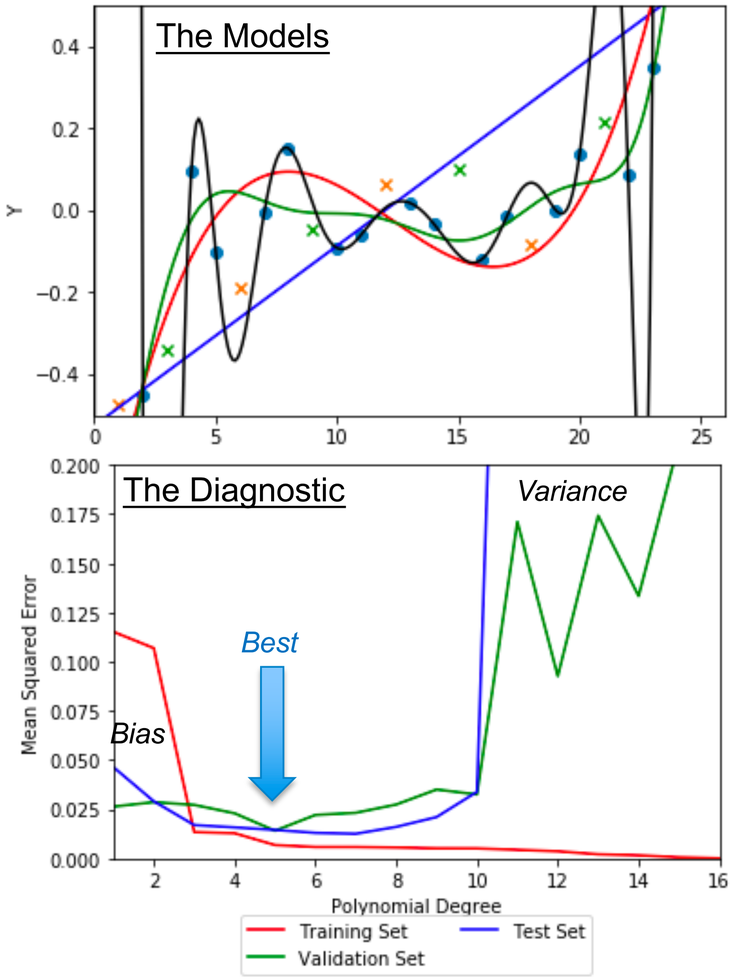


First sort the population into “strata”, <u>then</u> sample from each strata!

Validation Set

Example using different sets for polynomial fitting:

17 Training data
5 Validation data
5 Test Data



Using $J_{train}(\theta)$ and $J_{val}(\theta)$ for selecting the best regularization parameter:

The Training Set has m data points. The optimized Loss Function is (if L2 norm):

$$J(\theta) = \frac{1}{2m}\left(\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2 + \lambda \sum_{j=1}^{n} \theta_j^2\right)$$

For each tested value of λ, train the network on the Training Set, and evaluate the errors:

$$J_{train}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$$

And on the Validation Set:

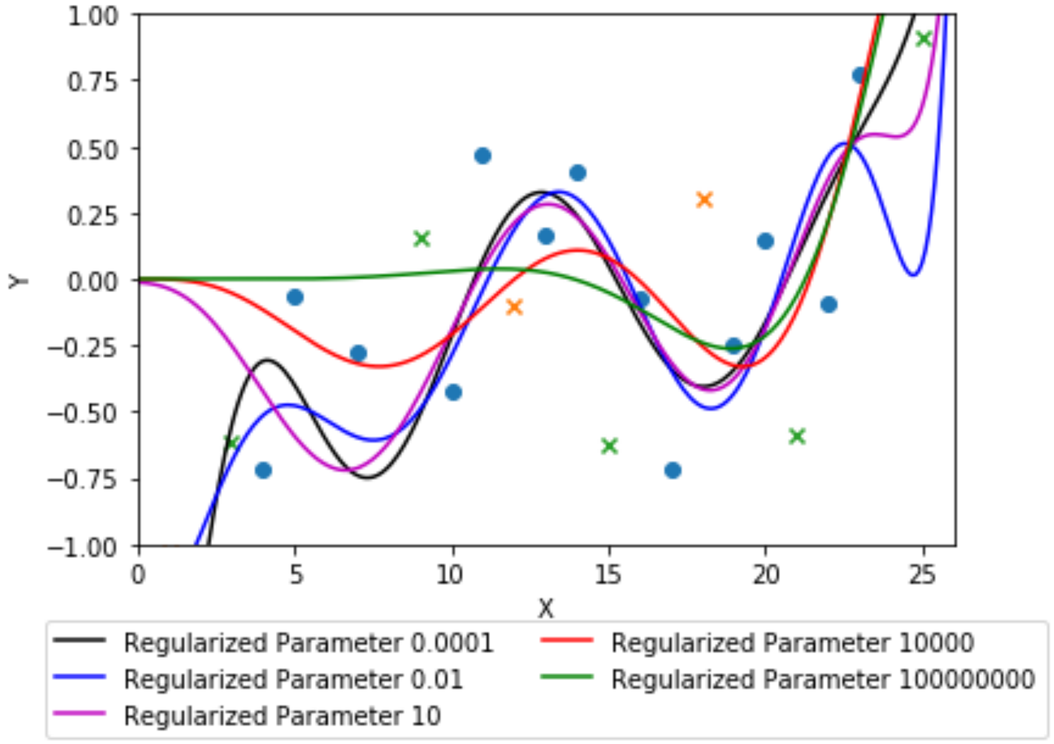
$$J_{val}(\theta) = \frac{1}{2m_{val}}\sum_{i=1}^{m_{val}}\left(h_\theta\left(x_{val}^{(i)}\right) - y_{val}^{(i)}\right)^2$$

On the Test Set:

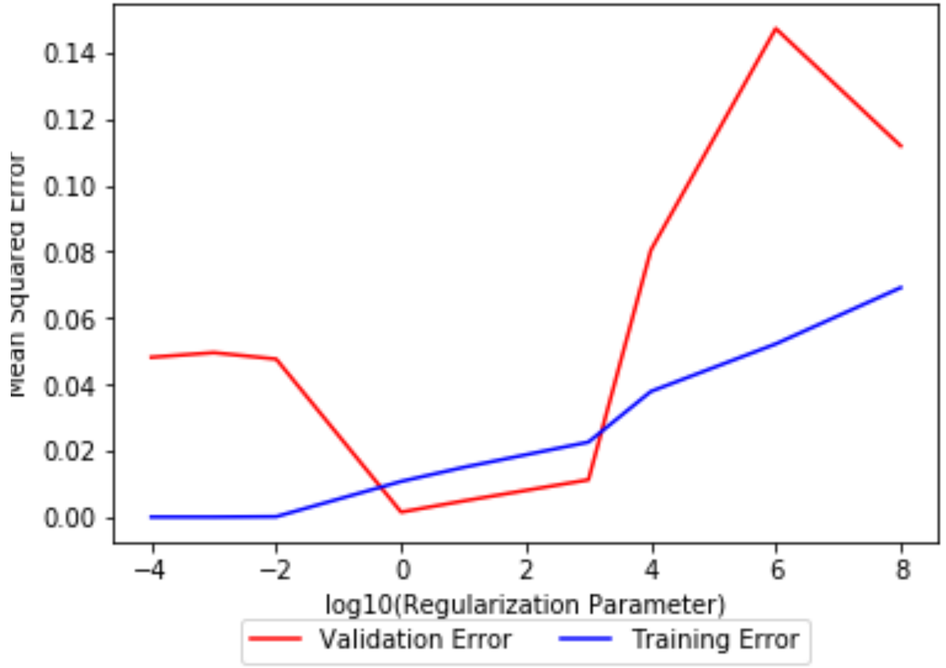
$$J_{test}(\theta) = \frac{1}{2m_{test}}\sum_{i=1}^{m_{test}}\left(h_\theta\left(x_{test}^{(i)}\right) - y_{test}^{(i)}\right)^2$$

Validation Set

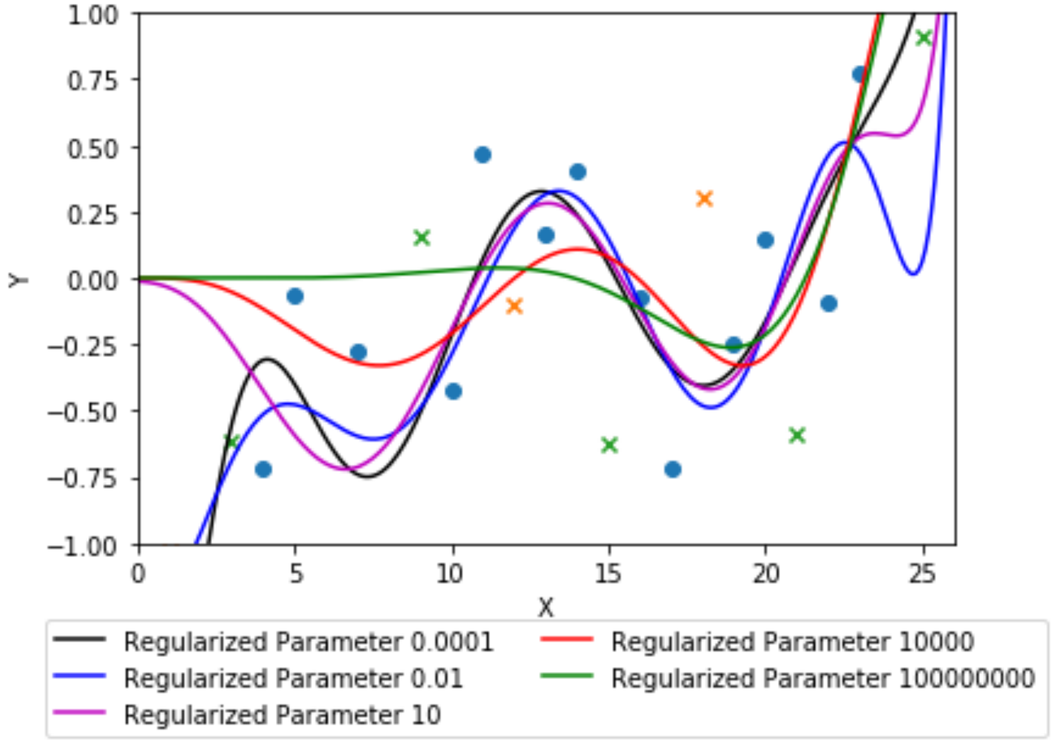
Optimising the regularization parameter:



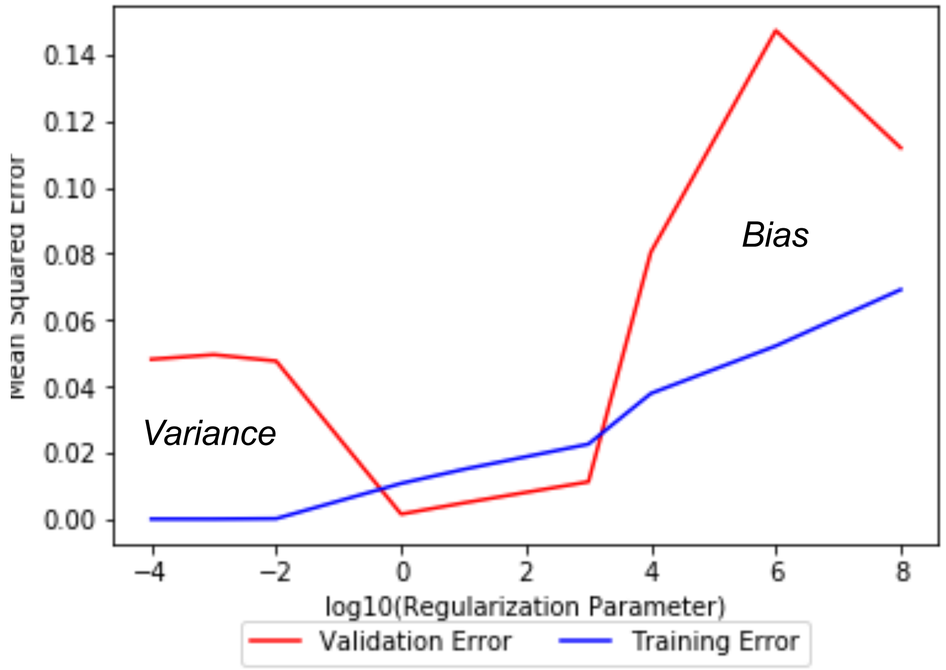
where is the bias and the variance? left of right?



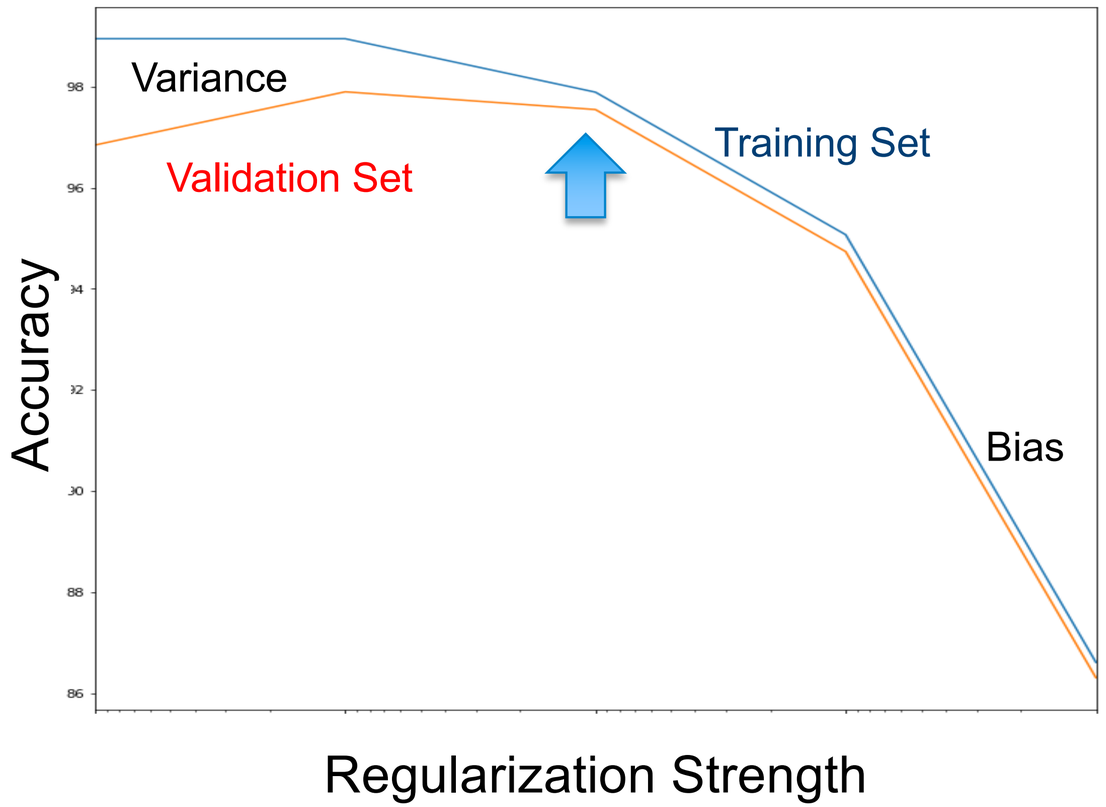
Optimising the regularization parameter:



where is the bias and the variance? left of right?



Typical regularization behaviour in classification:



Validation set on MNIST example:

The “official” MNIST Training Set is 60000 data. The “official” Test Set is 10000 and unknown to the user.

The user first needs to split the official Training Set into a new Training and Validation sets, for example with 50000 and 10000 data points (as many as in the Test Set) in each what kind of splitting should we do?

Then for a number of possible Hyperparameters:

1. Train Neural Network on Training Set
2. Test its Performance on Validation Set

Then pick the Hyperparameters that gives the optimal performance on the Validation Set. *Then possibly retrain the network on Training+Validation Sets.*

Then finally evaluate the performance of the Neural Network associated with these optimal Hyperparameters on the Test Set.

Validation set on MNIST example:

The “official” MNIST Training Set is 60000 data. The “official” Test Set is 10000 and unknown to the user.

The user first needs to split the official Training Set into a new Training and Validation sets, for example with 50000 and 10000 data points (as many as in the Test Set) in each (because of large number of data, Random Shuffling is enough).

Then for a number of possible Hyperparameters:

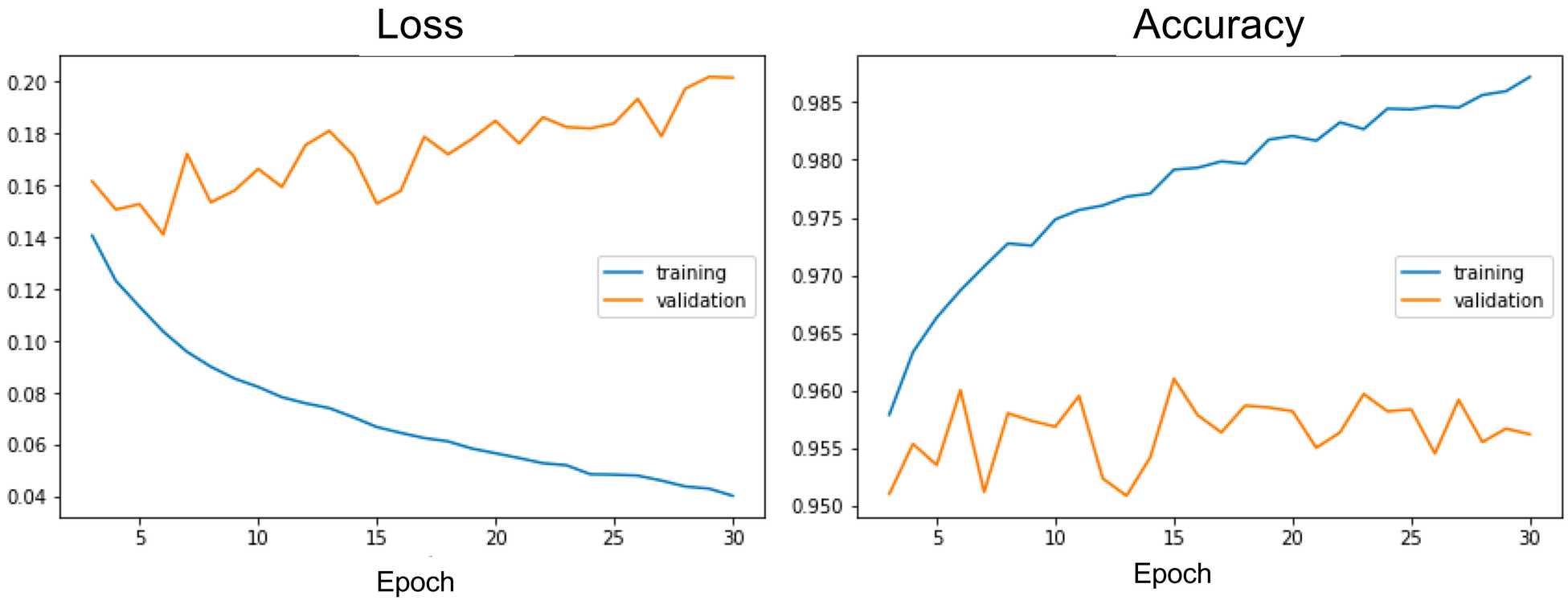
1. Train Neural Network on Training Set
2. Test its Performance on Validation Set

Then pick the Hyperparameters that gives the optimal performance on the Validation Set. *Then possibly retrain the network on Training+Validation Sets.*

Then finally evaluate the performance of the Neural Network associated with these optimal Hyperparameters on the Test Set.

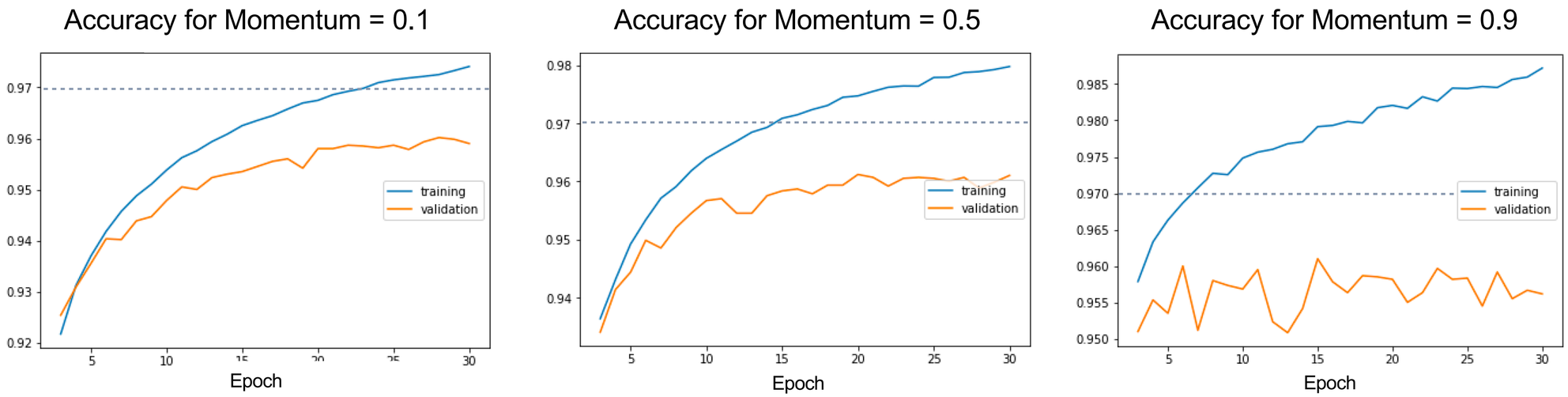
VALIDATION SET

Using validation set on **MNIST**:



is this training good?

Using validation set on MNIST:

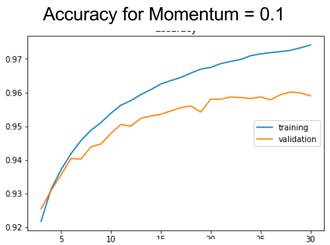


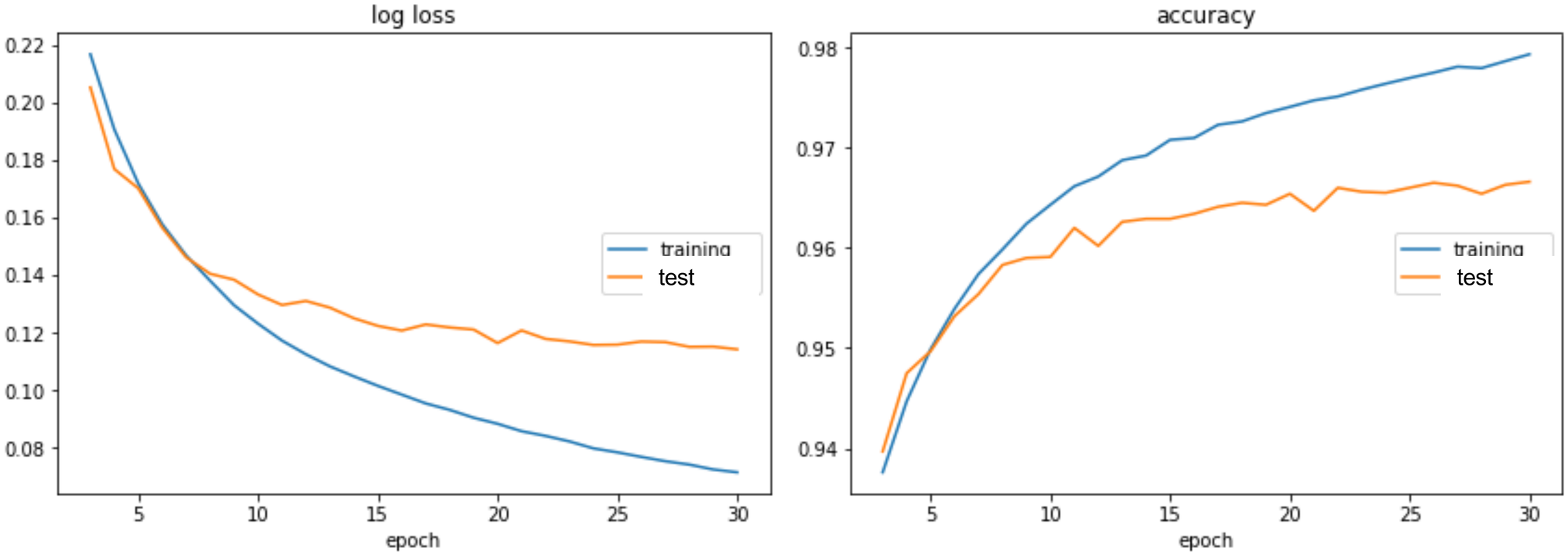
which of the three options is the best?

"Grid Search" evaluates Validation Accuracy for a set of Hyperparameter(s) values

VALIDATION SET

Final training on **MNIST** using training & validation sets:

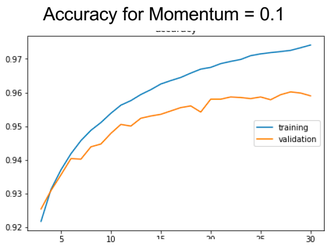


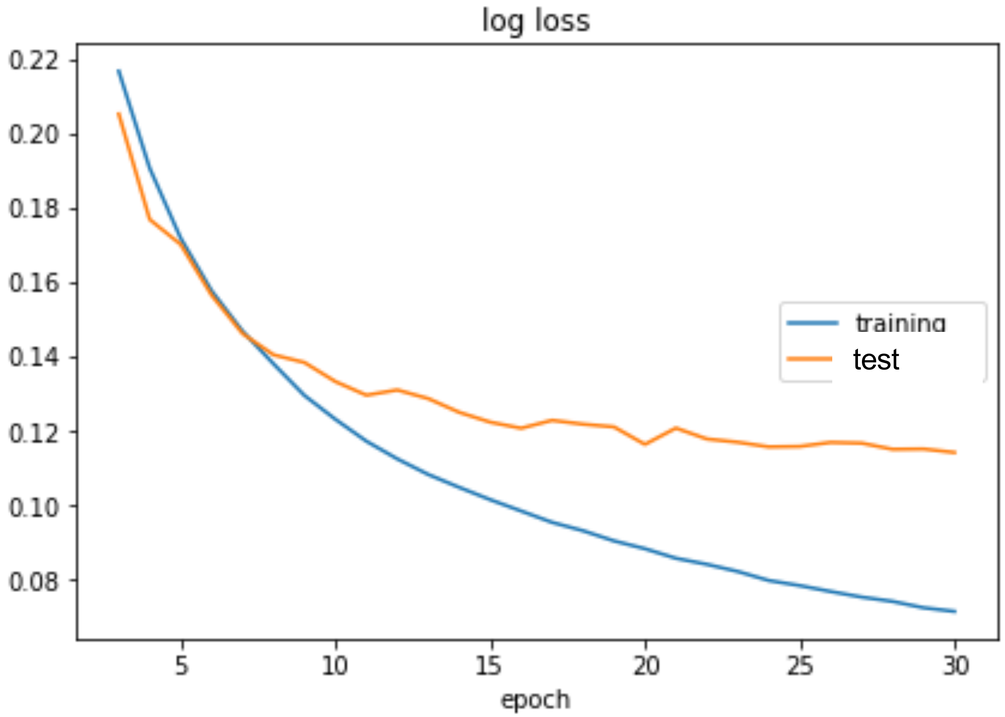


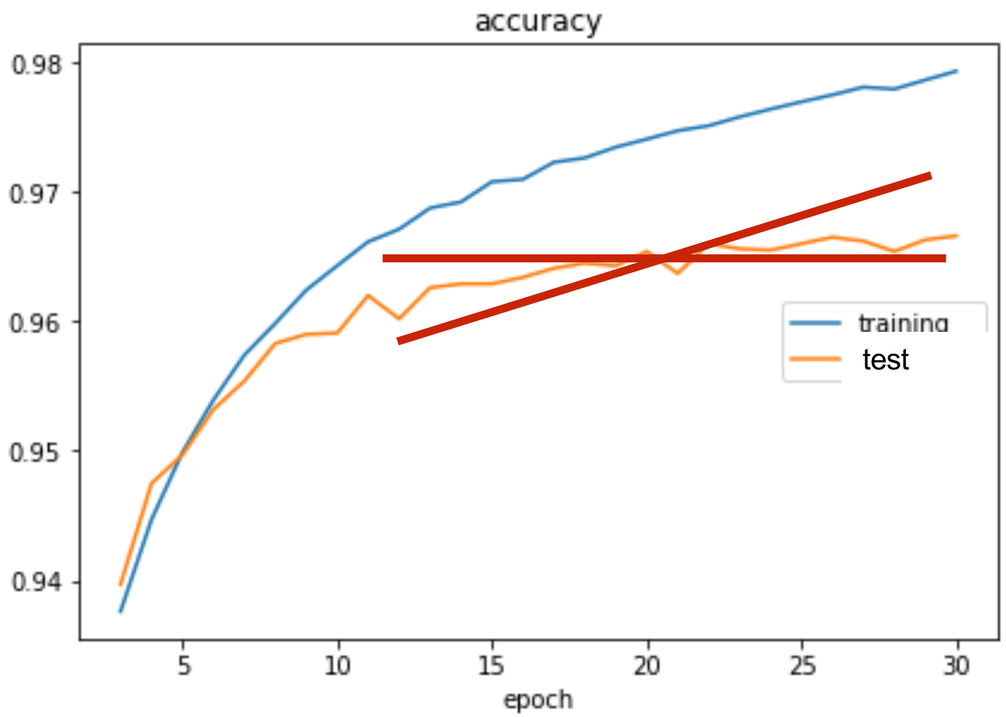
Test Set accuracy at about .97!

VALIDATION SET

Final training on **MNIST** using training & validation sets:

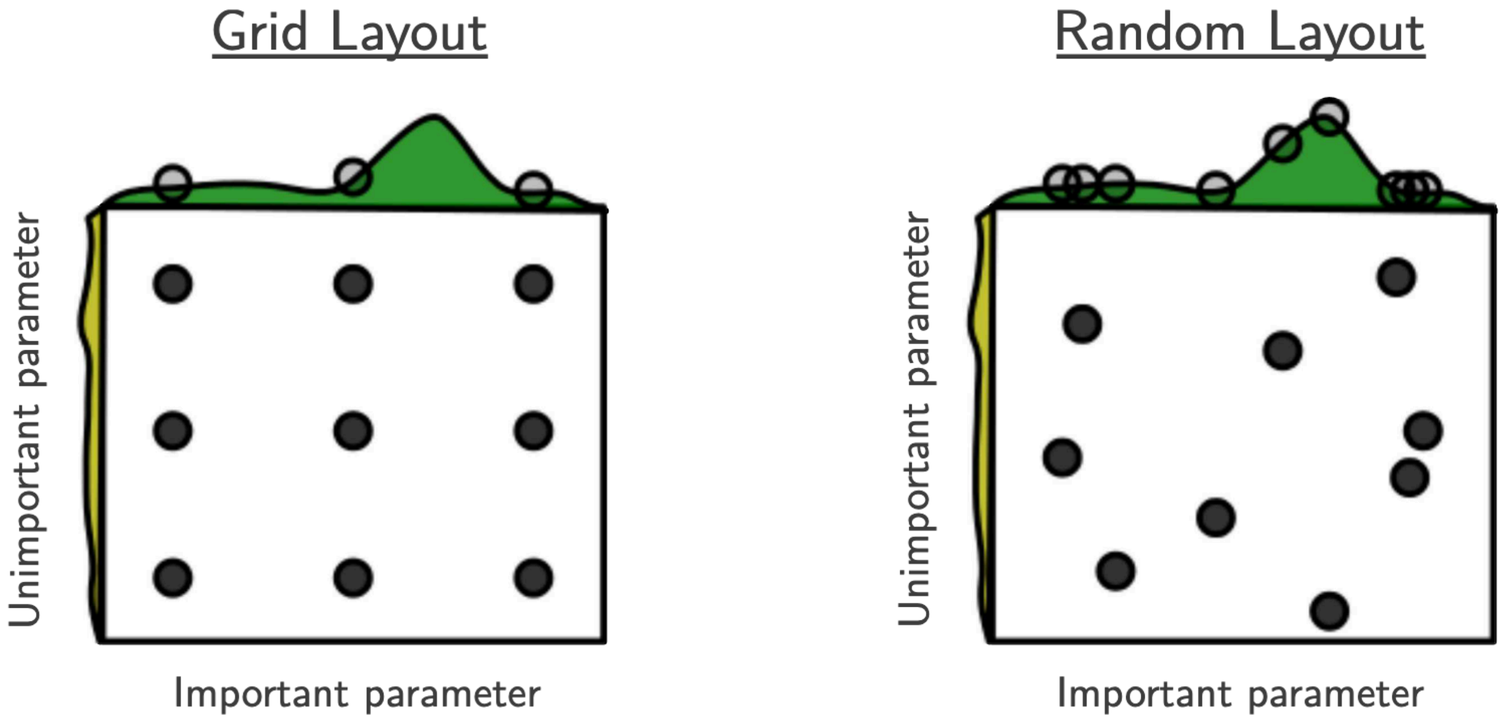






usually, we don't know accuracy for the test set

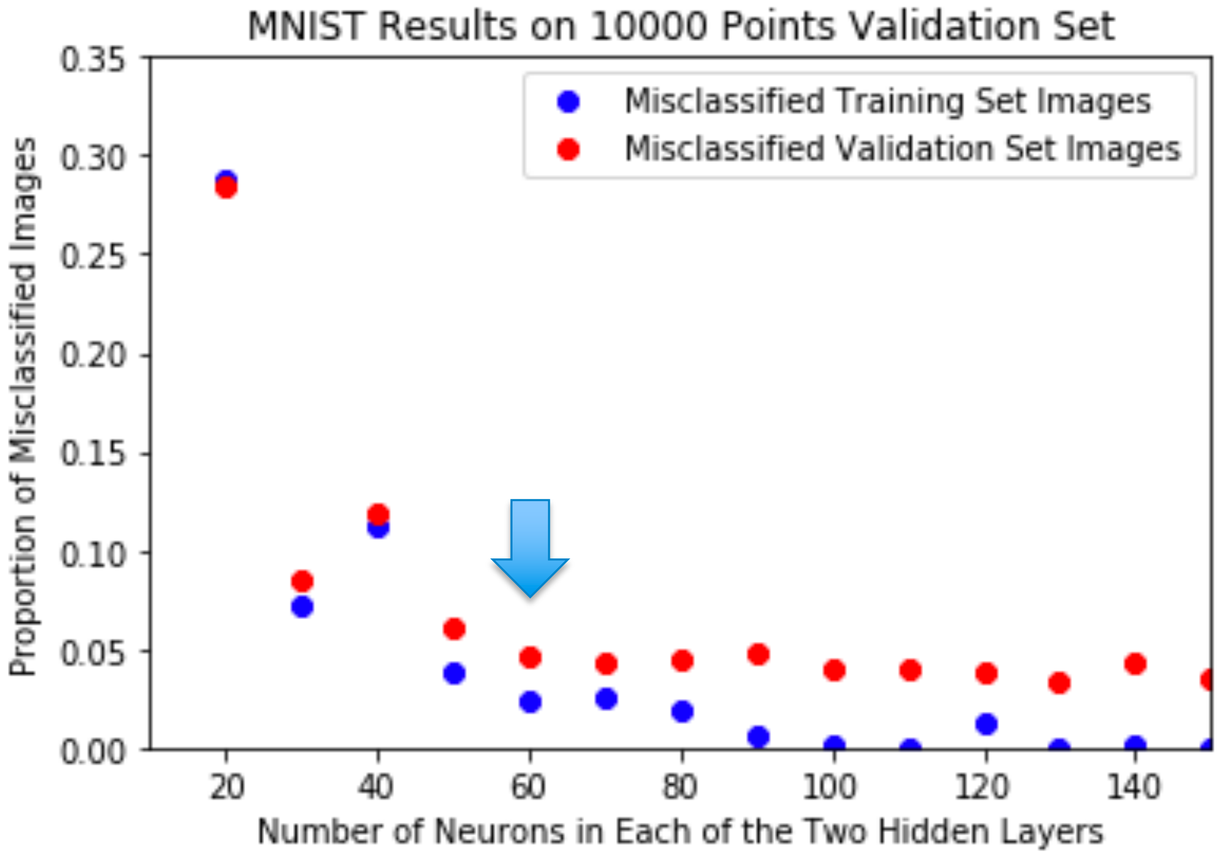
Grid search to optimise multiple hyperparameters:



https://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf

https://www.youtube.com/watch?v=AXDByU3D1hA

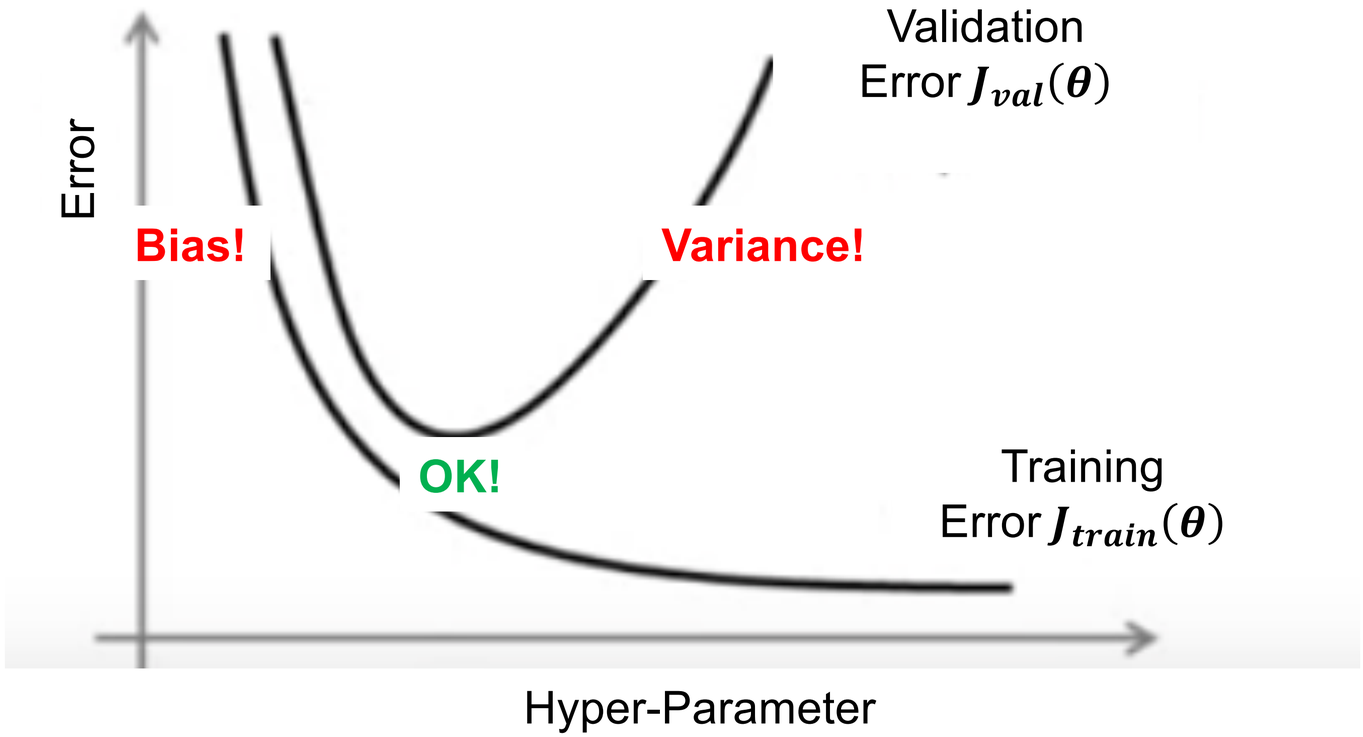
MNIST: optimising number of neurons in hidden layers



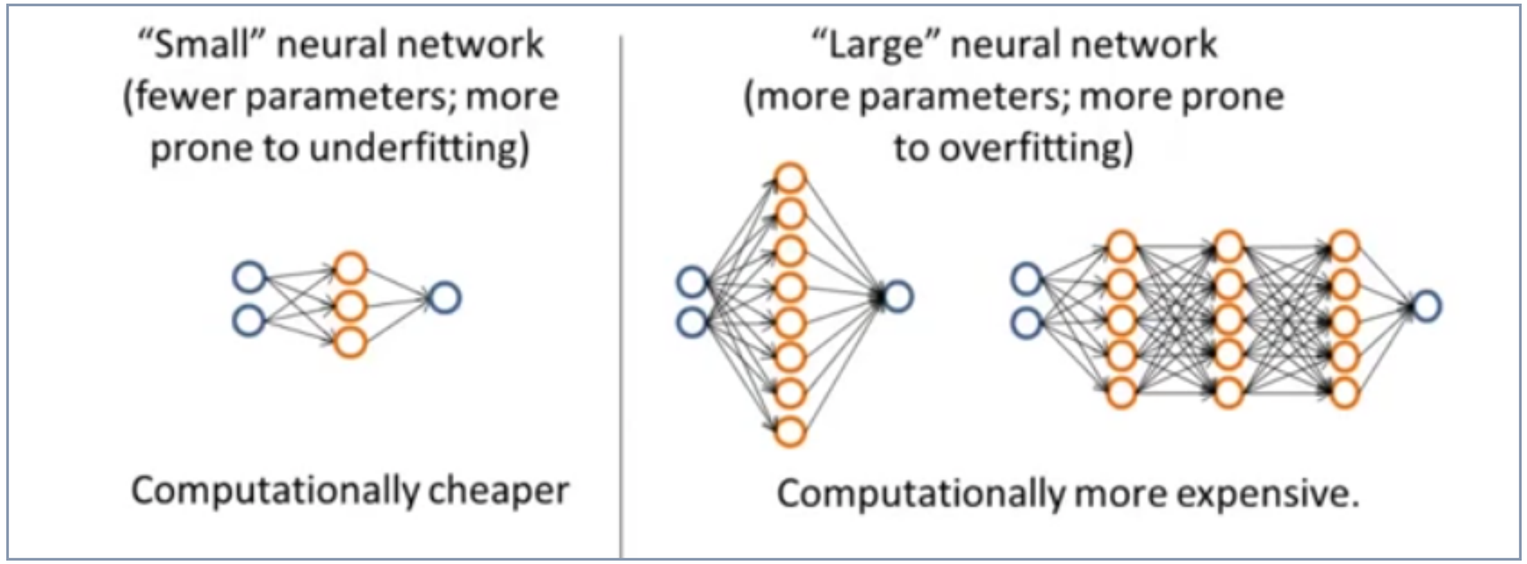
Optimal value seems to be about 60-70 neurons in each of the two hidden layers.

This hyperparameter value gives a proportion of 0.04 misclassified images in the Test Set (same as Validation Set!)

Identifying **bias** and **variance** problems:



Bias/underfitting and variance/overfitting in networks:



General guidelines to improve training:

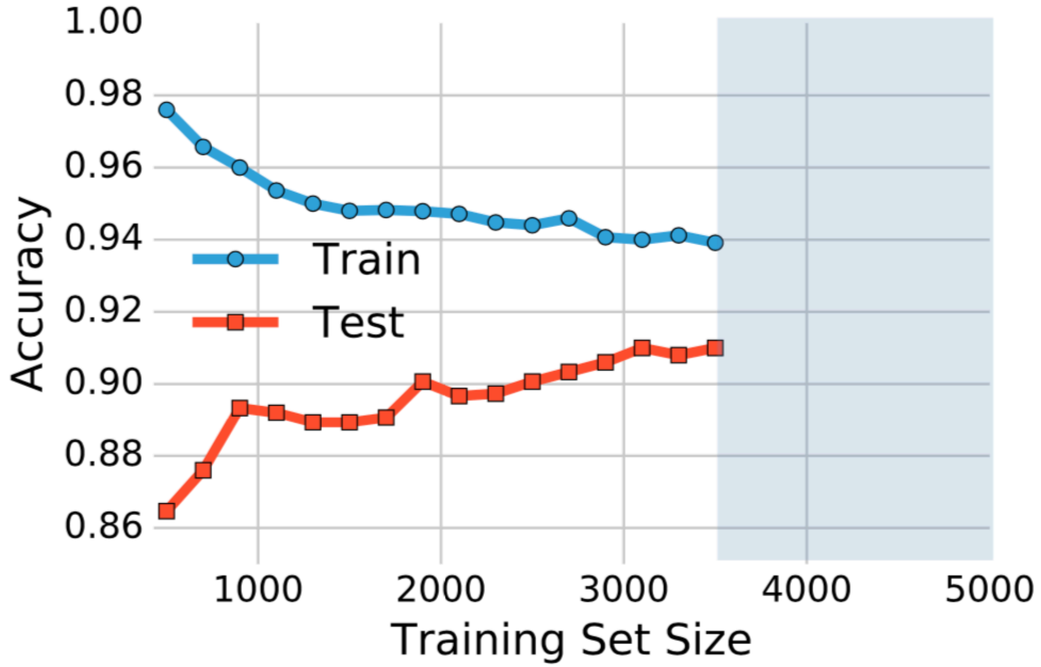
To **address bias** problems:

- Increase number of samples in the data (if you have it)
- Increase number of parameters in the network
- Decrease regularisation

To address variance problems:

- Increase number of samples in the data (if you have it) - *more data is (almost) always better*
- Decrease number of parameters in the network
- Increase regularisation

Impact of number of training samples in MNIST:



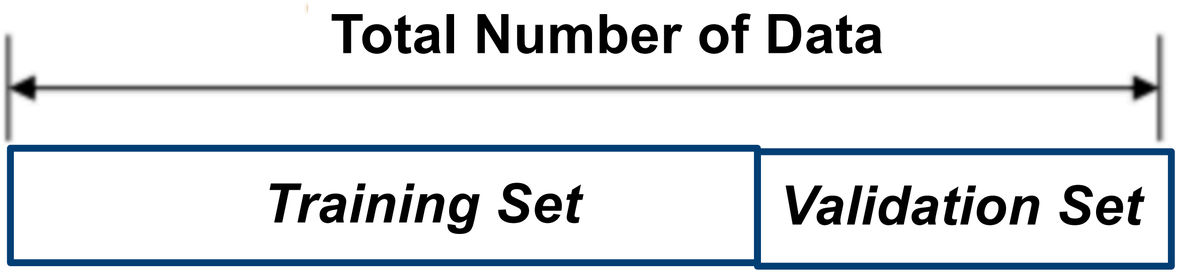
The larger the number of Training Data, the less chance of Overfitting!

Regularisation, Bias, and Variance

1. Overfitting and underfitting, bias and variance
2. Regularization
3. Batch normalization
4. Machine learning diagnostics
5. Training, validation, and test sets
6. **K-fold validation**

Holdout method: use only one validation set

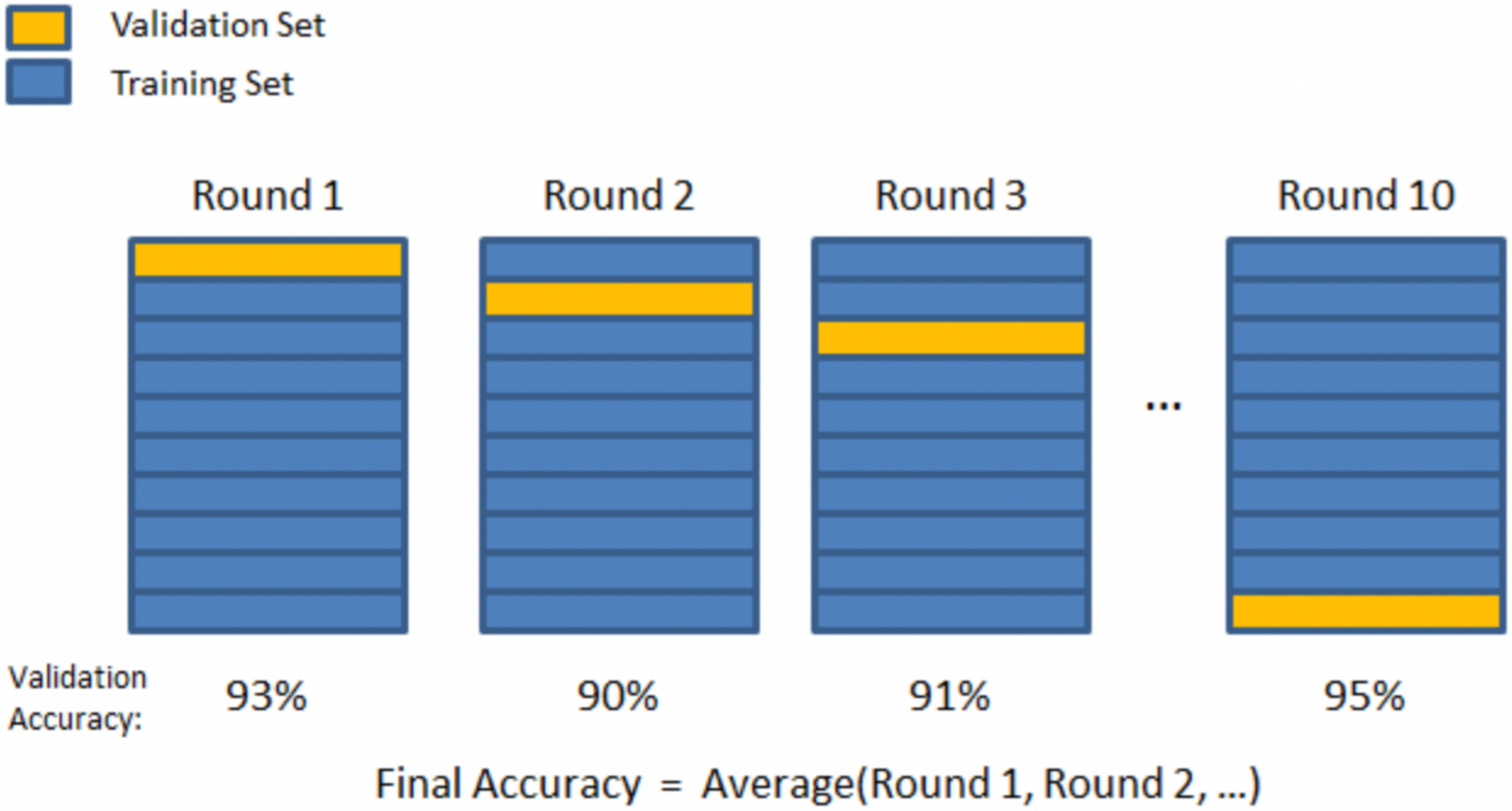
If Data Set is not very big, the role played by Training and Validation sets is not symmetrical. The approach used for the split may significantly affect the results.



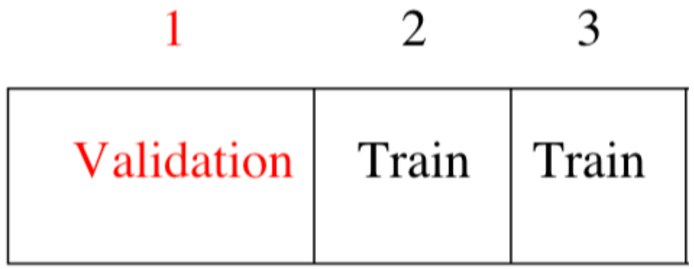
Possible approach: permutate between Validation and Training Sets and recalculate. Can do it if size of both is similar.

Multiple Validation Sets

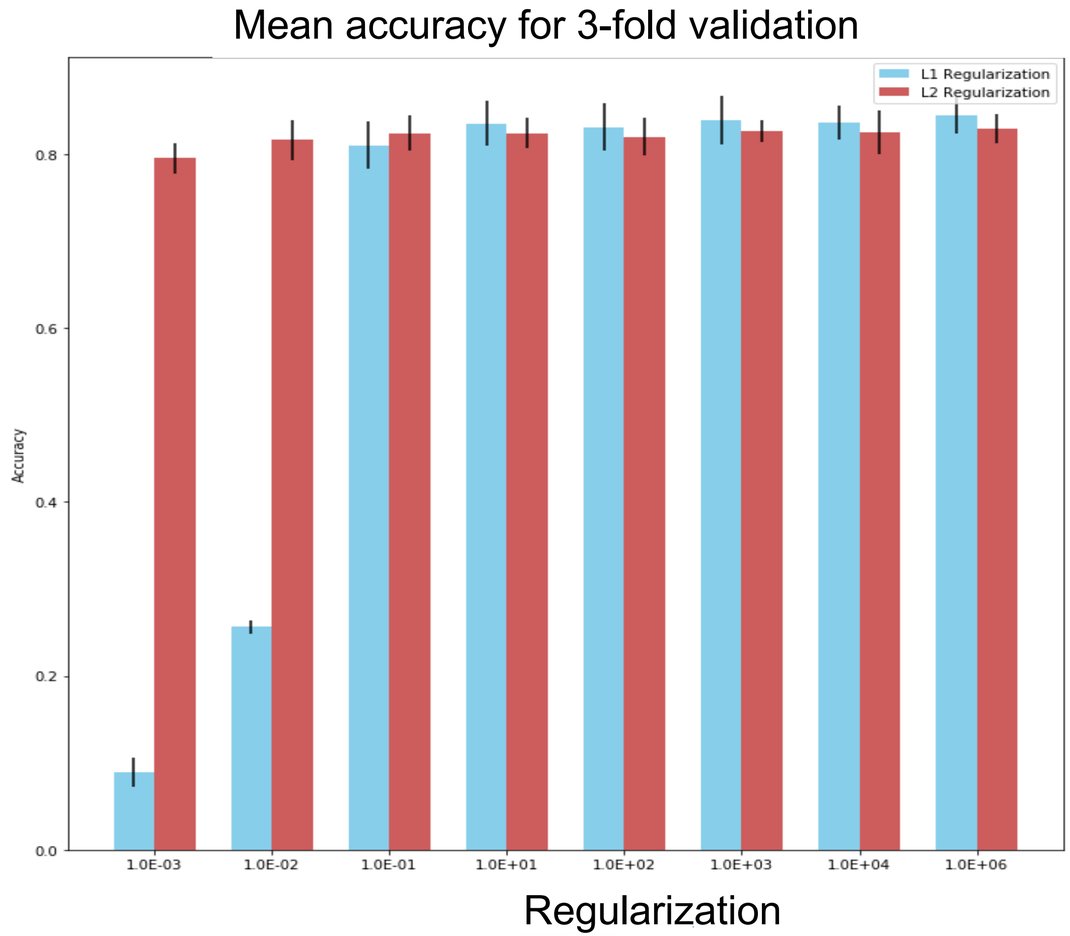
k-fold validation: use multiple validation sets (without replacement)



3-fold validation example with MNIST



For each value of the Regularization Parameter, a 3-fold validation is run: three validation sets are created in turn and predicted using the rest of the data as training set.



Afternoon exercise:

Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning

Sebastian Raschka
University of Wisconsin–Madison
Department of Statistics
November 2018
sraschka@wisc.edu

Abstract

The correct use of model evaluation, model selection, and algorithm selection techniques is vital in academic machine learning research as well as in many industrial settings. This article reviews different techniques that can be used for each of these three subtasks and discusses the main advantages and disadvantages of each technique with references to theoretical and empirical studies. Further, recommendations are given to encourage best yet feasible practices in research and applications of machine learning. Common methods such as the holdout method for model evaluation and selection are covered, which are not recommended when working with small datasets. Different flavors of the bootstrap technique are introduced for estimating the uncertainty of performance estimates, as an alternative to confidence intervals via normal approximation if bootstrapping is computationally feasible. Common cross-validation techniques such as leave-one-out cross-validation and k-fold cross-validation are reviewed, the bias-variance trade-off for choosing k is discussed, and practical tips for the optimal choice of k are given based on empirical evidence. Different statistical tests for algorithm comparisons are presented, and strategies for dealing with multiple comparisons such as omnibus tests and multiple-comparison corrections are discussed. Finally, alternative methods for algorithm selection, such as the combined F-test 5x2 cross-validation and nested cross-validation, are recommended for comparing machine learning algorithms when datasets are small.

Summary

- Bias/variance or underfitting/overfitting.
- Regularisations to reduce overfitting: L1 & L2 examples.
- Dropout, data augmentation, and other regularisers.
- Batch normalization to accelerate training and as a regulariser.
- Accuracy metrics.
- Importance of training, validation and test set in ML diagnostics.
- K-Fold validation for hyperparameter tuning.