The How, What and Why of Deep Learning

Seminar 2: Training Neural Networks

Backpropagation, Stochastic Gradient Descent and Regularisation

Big Data Institute, Oxford
April 5th, 2019

Alexander Camuto & Matthew Willetts

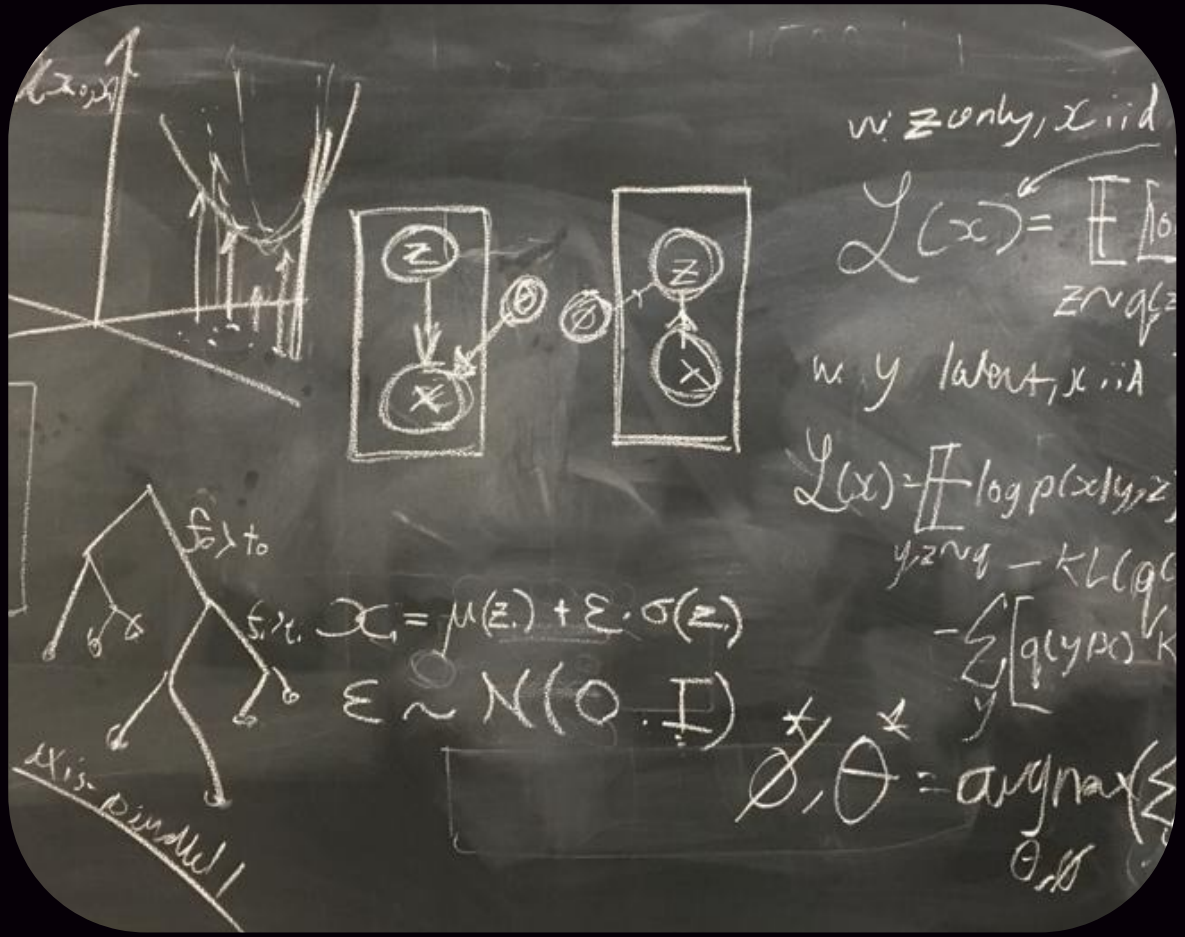
Course Timetable

- **22nd March**: Introduction: MLPs

***5th April**: Computational Graphs: Implementations with Keras, Optimisation & Regularisation methods*

***5th April**: Computational Graphs: Implementations with Keras, Optimisation & Regularisation methods*

- **19th April**: Convolutional NNs
- **3rd May**: RNNs and LSTMs
- **17th May:** Auto Encoders and NLP
- **31st May:** Deep Generative Models: VAEs & GANs, and their implementation with Tensorflow
- **14th June:** Deep RL & Frontiers in Deep Learning



Recap and Today

Last Time

- Multilayer Perceptrons are stacked Logistic Regression
- We have an objective (aka cost or loss) to optimise
- Learning is adjusting the parameters of our model so it does well under this objective
- We do this by taking gradients of the objective wrt all model parameters, then stepping the parameters in that direction to optimise that objective.

TODAY

- How to do this, but for real though

Backpropagation: Simplified network

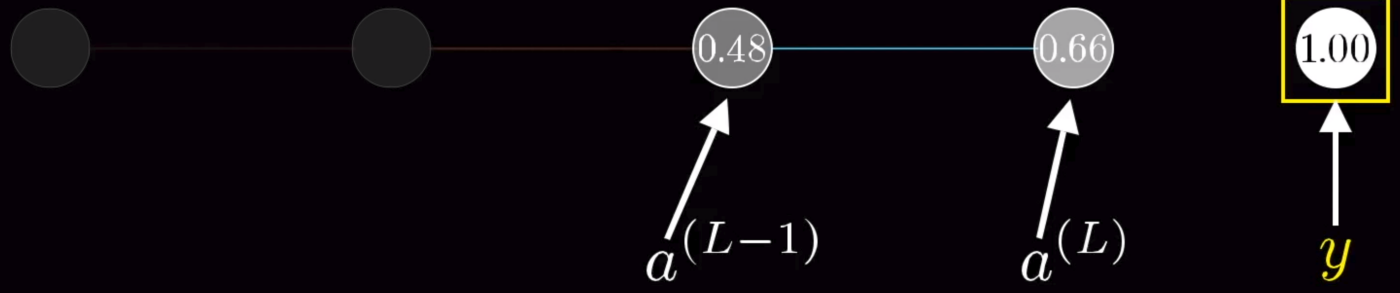
$$C(w_1, b_1, w_2, b_2, w_3, b_3)$$

Backpropagation: What are our terms

$$\text{Cost} \longrightarrow C_0(\dots) = (a^{(L)} - y)^2$$

$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

$$a^{(L)} = \sigma(z^{(L)})$$

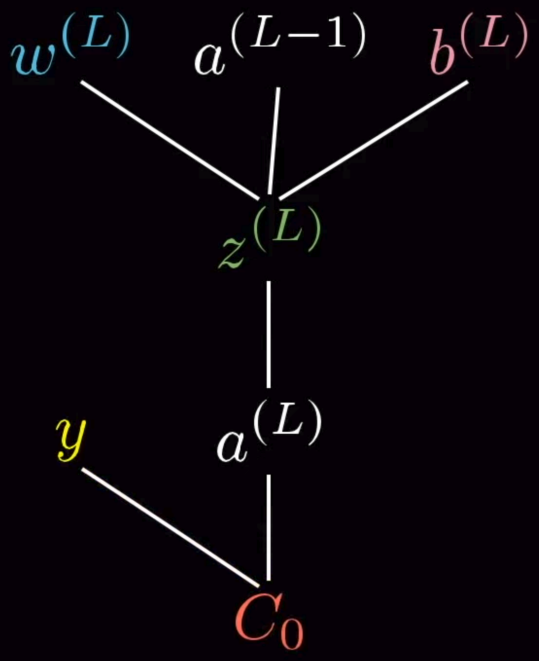


Backpropagation: view computation as a graph

$$\text{Cost} \longrightarrow C_0(\dots) = (a^{(L)} - y)^2$$

$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

$$a^{(L)} = \sigma(z^{(L)})$$



Desired output

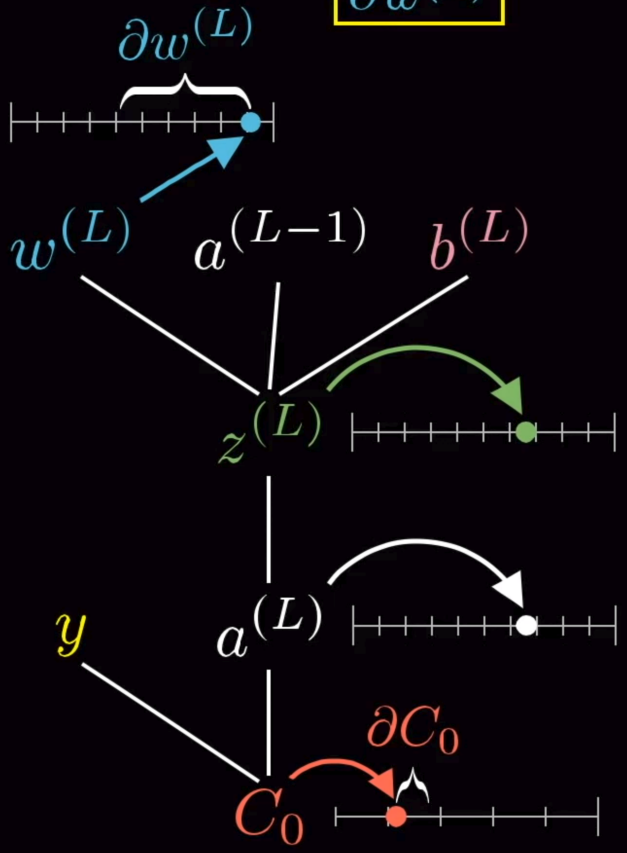
Backpropagation: Actually taking gradients

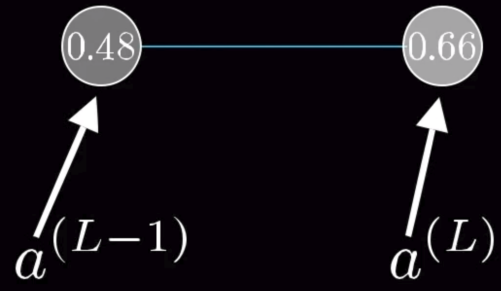
$\frac{\partial C_0}{\partial w^{(L)}}$ What we want

$C_0(\dots) = (a^{(L)} - y)^2$

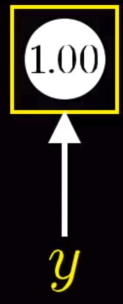
$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$

$a^{(L)} = \sigma(z^{(L)})$





Desired output



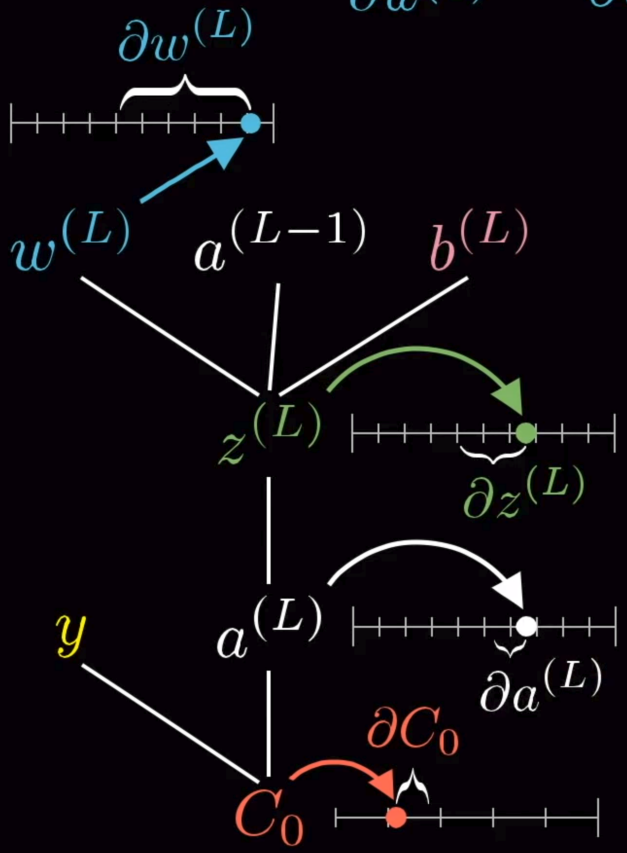
Backpropagation: Apply Chain Rule

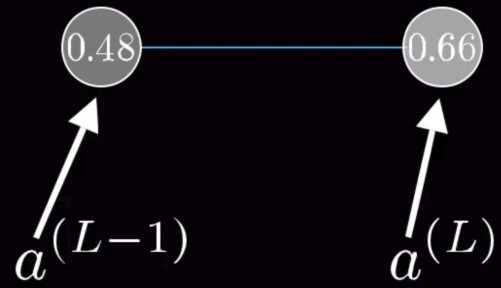
$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C0}{\partial a^{(L)}}$$

$$C_0(\dots) = (a^{(L)} - y)^2$$

$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

$$a^{(L)} = \sigma(z^{(L)})$$





Desired output

Backpropagation: Apply Chain Rule

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C0}{\partial a^{(L)}}$$

$$\frac{\partial C0}{\partial a^{(L)}} = 2(a^{(L)} - y)$$

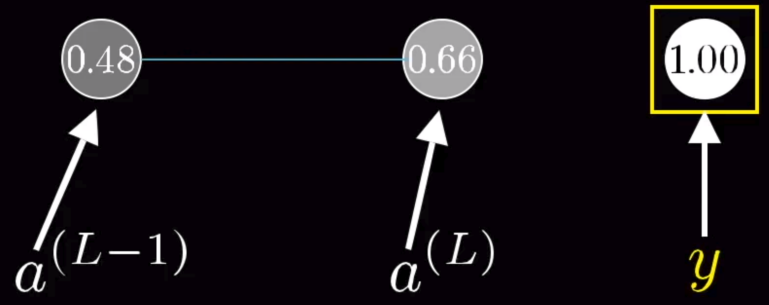
$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'(z^{(L)})$$

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$$

$$C_0 = (a^{(L)} - y)^2$$

$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

$$a^{(L)} = \sigma(z^{(L)})$$



Backpropagation: Summing

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C0}{\partial a^{(L)}} = a^{(L-1)} \sigma'(z^{(L)}) 2(a^{(L)} - y)$$

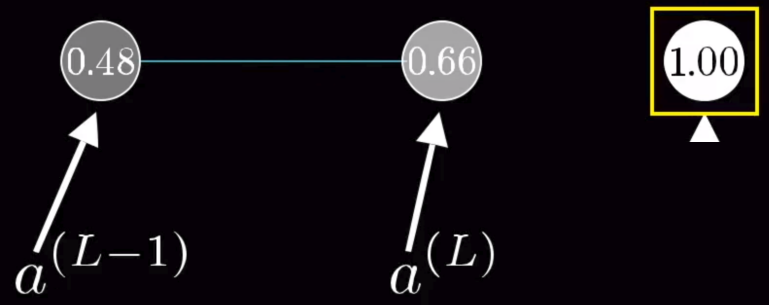
$$C_0 = (a^{(L)} - y)^2$$

$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

$$a^{(L)} = \sigma(z^{(L)})$$

Average of all training examples

$$\frac{\partial C}{\partial w^{(L)}} = \overbrace{\frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial C_k}{\partial w^{(L)}}}$$



Backpropagation: But what about other variables?

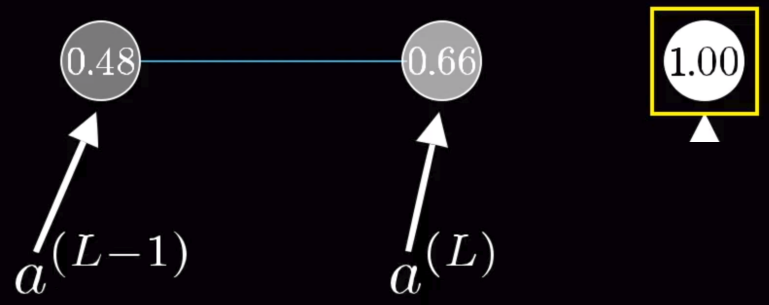
$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C0}{\partial a^{(L)}} = a^{(L-1)} \sigma'(z^{(L)}) 2(a^{(L)} - y)$$

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w^{(1)}} \\ \frac{\partial C}{\partial b^{(1)}} \\ \vdots \\ \frac{\partial C}{\partial w^{(L)}} \\ \frac{\partial C}{\partial b^{(L)}} \end{bmatrix}$$

$$C_0 = (a^{(L)} - y)^2$$

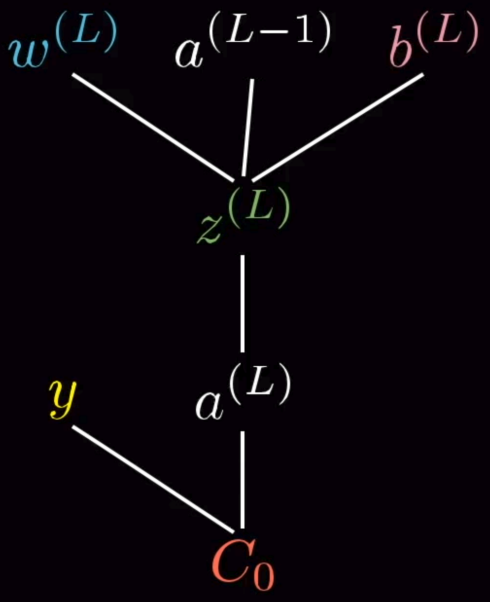
$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

$$a^{(L)} = \sigma(z^{(L)})$$



Backpropagation: Caching to calc other values

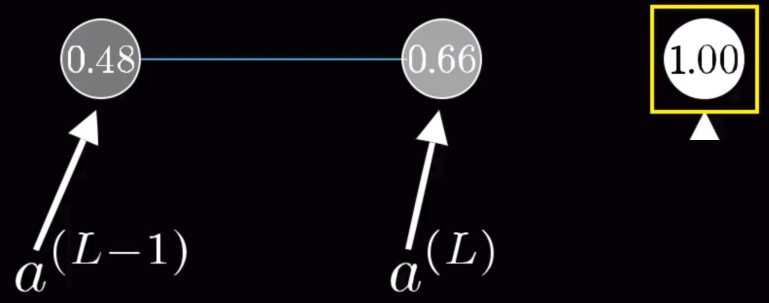
$$\frac{\partial C_0}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C0}{\partial a^{(L)}} = 1\sigma'(z^{(L)})2(a^{(L)} - y)$$



$$C_0 = (a^{(L)} - y)^2$$

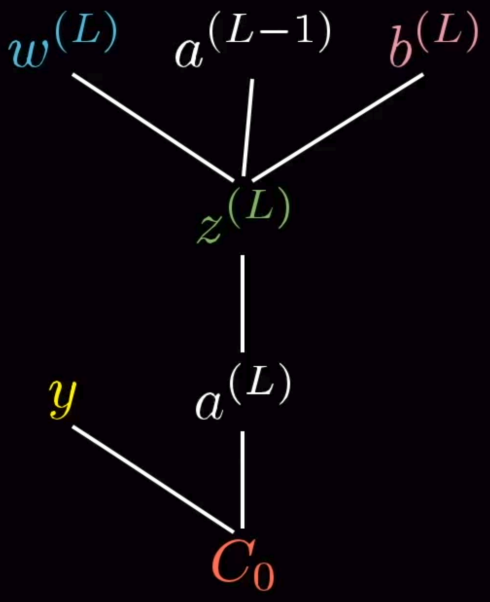
$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

$$a^{(L)} = \sigma(z^{(L)})$$



Backpropagation: And Backwards through layers

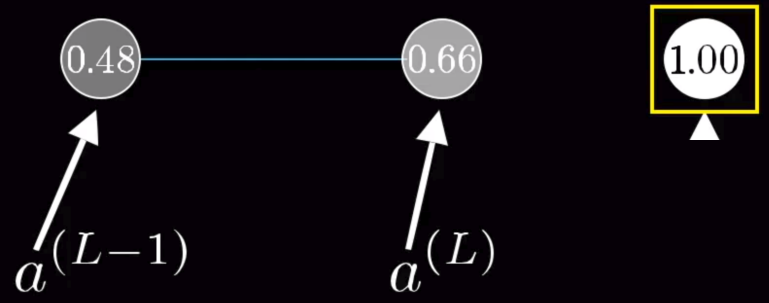
$$\frac{\partial C_0}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C0}{\partial a^{(L)}} = w^{(L)} \sigma'(z^{(L)}) 2(a^{(L)} - y)$$



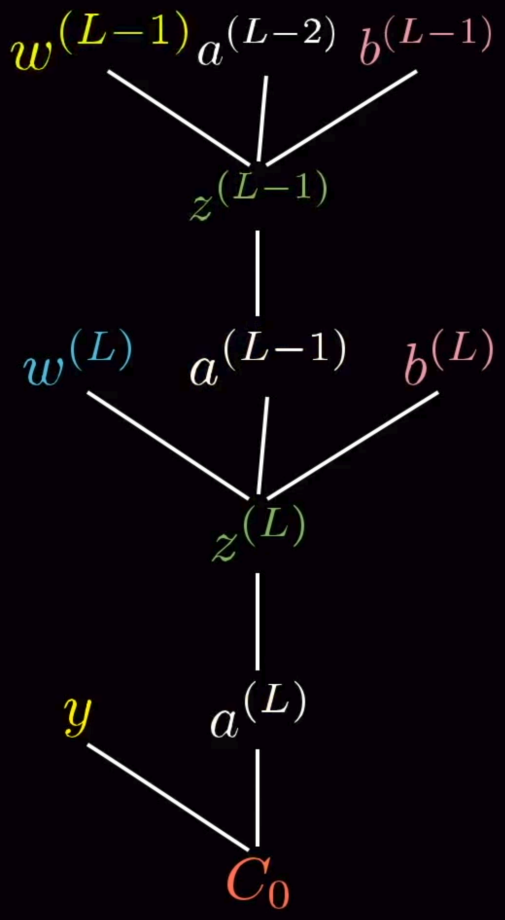
$$C_0 = (a^{(L)} - y)^2$$

$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

$$a^{(L)} = \sigma(z^{(L)})$$



Backpropagation: And Backwards through layers

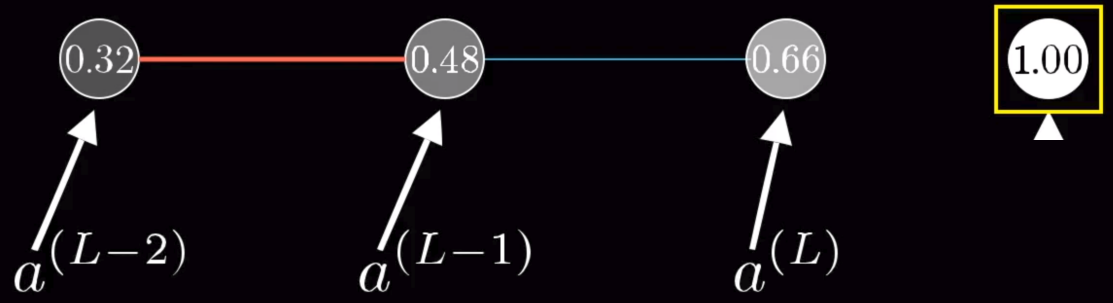


$$\frac{\partial C_0}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C0}{\partial a^{(L)}}$$

$$C_0 = (a^{(L)} - y)^2$$

$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

$$a^{(L)} = \sigma(z^{(L)})$$



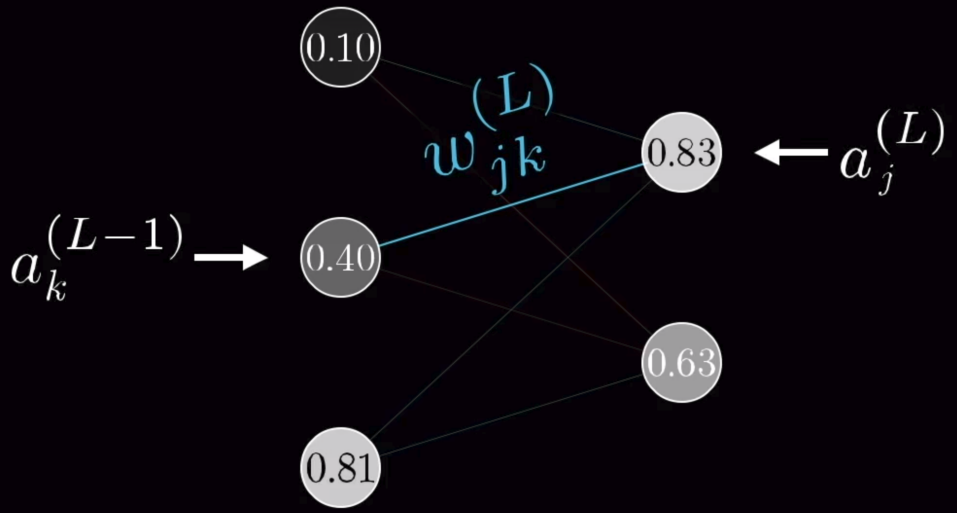
Backpropagation: More than one unit per layer

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}}$$

$$z_j^{(L)} = \cdots + w_{jk}^{(L)} a_k^{(L-1)} + \cdots$$

$$a_j^{(L)} = \sigma(z_j^{(L)})$$

$$C_0 = \sum_{j=0}^{n_L - 1} (a_j^{(L)} - y_j)^2$$



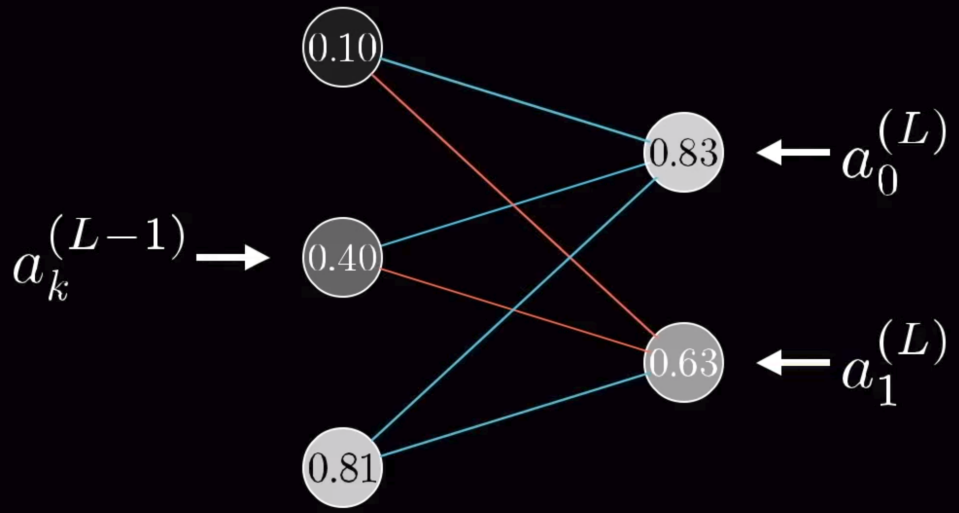
Backpropagation: More than one unit per layer

$$\frac{\partial C_0}{\partial a_k^{(L-1)}} = \underbrace{\sum_{j=0}^{n_L-1} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}}}_{\text{Sum over layer L}}$$

$$z_j^{(L)} = \cdots + w_{jk}^{(L)} a_k^{(L-1)} + \cdots$$

$$a_j^{(L)} = \sigma(z_j^{(L)})$$

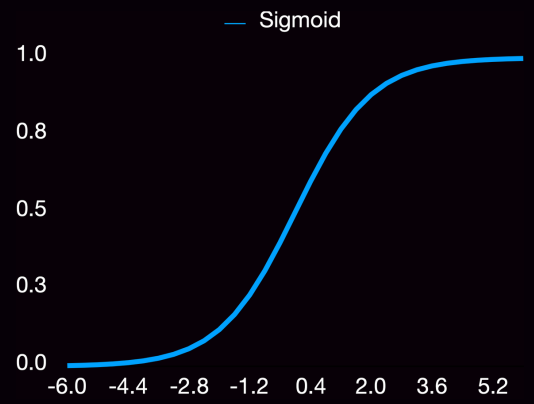
$$C_0 = \sum_{j=0}^{n_L-1} (a_j^{(L)} - y_j)^2$$

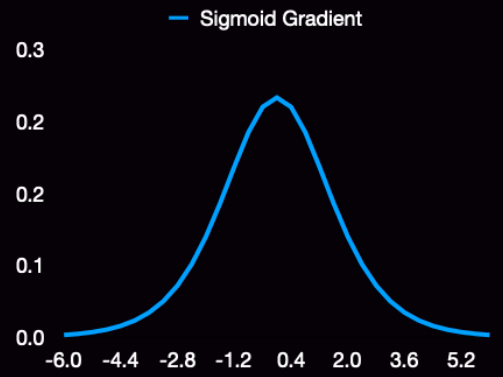


Why ReLUs, again

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}}$$

$$a_j^{(L)} = \sigma(z_j^{(L)})$$





If input to sigmoid is large, middle term in gradient above goes to zero

Thus the gradient wrt parameters goes to zero

But if activation is ReLU, middle term in gradient = 1 for z>0

Stoch Grad - Why we minibatch

$$\frac{\partial C}{\partial w^{(L)}} = \overbrace{\frac{1}{n}\sum_{k=0}^{n-1} \frac{\partial C_k}{\partial w^{(L)}}}^{\text{Average of all training examples}}$$

Often too large to fit all terms in sum in memory all at once: we would have to keep all the results of computation for all data in dataset

Dataset that itself may be Tb in size: with model params + grads this is huge

So instead need something more like streaming

Get estimate of gradient by getting a subsample of data to train on for each step

Extreme approach: take one datapoint at random and take step from grad eval on that.
Downsides: High variance in grad estimate, doesn't use full parallelism of modern hardware

Reasonable approach: sample some 'minibatch' of something like 128, 256, 512 datapoints in each training step
(Powers of 2 for hardware - can be an order of mag faster than batch size of, say, 100)

SGD - Stochastic Gradient - Again

Writing out the maths: The correct 1st order gradient descent is:

$$\theta_{t+1} = \theta_t - \rho(t)\nabla_\theta(\frac{1}{n}\sum_i^n C(\mathbf{x_i}, y; \theta))|_{\theta=\theta_t}$$

But we cannot fit all of our data in memory to do this sum, so as discussed:
→ sample a 'minibatch' of n_bs datapoints without replacement

$$\theta_{t+1} = \theta_t - \rho(t)\nabla_\theta(\frac{1}{n_{bs}}\sum_i^{n_{bs}} C(\mathbf{x_i}, y; \theta))|_{\theta=\theta_t}$$

$$\text{where } i \sim \{1, ..., n\}$$

- Resample the minibatch indices each training step
- *NOTE: We obtain noisy (but unbiased) estimates of gradient*
  - But that might be a good thing - noise means we can escape local minima
- Common to spend time optimising the ρ(t) values - known as the “annealing schedule”

Stoch Grad - Why we minibatch

Recall:

In Non-Convex Optimisation, the Local Minima you reach will be determined by the objective function you have AND the optimisation routine used

We find that smallish batch sizes lead to models with better generalisation - we end up in broadered, shallower minima due to increased stochasticity of gradients, and these seem to be good ones to end up in. *See Zhang et al.*

Stoch Grad - But How

Recall:

In Non-Convex Optimisation, the Local Minima you reach will be determined by the objective function you have AND the optimisation routine used

We have lots of design choices available for how to traverse these loss landscapes. We've already seen SGD numerous times.
Overview two commom algos with adaptive learning Rates:
RMSprop & ADAM

Adaptive Learning Rate Methods

For conciseness, denote the loss over one sampled minibatch as $C(\theta) = \frac{1}{n_{bs}} \sum_{i}^{n_{bs}} C(\mathbf{x_i}, y; \theta)$

We can end up with different units receiving gradient steps of very different magnitudes

Recall from Backprop, sum over connected units. So unit with large 'fan in' has large gradient

How can we have a different learning rate then for different directions in θ?

Hack if taking grads over whole-dataset - move the same distance in all directions θ:
--> throw away magnitude of ∇C, just use its sign.

But how about for minibatch-based gradient decent algorithms?
How can we have a seperate learning rate for each direction?

RMS Prop

$$\eta_{t+1} = \alpha\eta_t + (1-\alpha)||\nabla_\theta C(\theta)|_{\theta=\theta_t}||^2$$

$$\Delta\theta_t = -\frac{\rho_0}{\sqrt{\eta_{t+1}+\epsilon}} \cdot \nabla_\theta C(\theta)|_{\theta=\theta_t}$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Here we keep a running (exponentially decaying) average of the square of the gradient for each point.

1/sqrt of this is our local learning rate (times ρ_0, our base learning rate)

We then use this with as our local step size for each gradient update

Commonly α=0.9 and ρ_0=0.01, but do tune ---- hyperoptimisation. ε=10e-7

ADAM

First recall Momentum from last time

1. We decay our velocity v and add a bit of acceleration
2. Then move in the direction of our new velocitv vector

$$v_{t+1} = \alpha v_t - \epsilon \nabla_\theta C(\theta)|_{\theta=\theta_t}$$
$$\Delta\theta_t = v_{t+1}$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

And now with RMSprop we

1. We decay our per-dim learning rate memory η and add a downweighted square of the gradient
2. Use this to calc a new per-dim learning rate, using the sqrt of η
3. Then move using our per-dim lr-weighted gradient

$$\eta_{t+1} = \alpha\eta_t + (1-\alpha)||\nabla_\theta C(\theta)|_{\theta=\theta_t}||^2$$
$$\Delta\theta_t = -\frac{\rho_0}{\sqrt{\eta_{t+1}+\epsilon}} \cdot \nabla_\theta C(\theta)|_{\theta=\theta_t}$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

ADAM: DO BOTH

ADAM

V(t) updates very similar to momentum, but now

convex combination

η same as RMSprop

Then scale a little, and do both!

$$v_{t+1} = \beta_1 v_t + (1-\beta_1)\nabla_\theta C(\theta)|_{\theta=\theta_t}$$

$$\eta_{t+1} = \beta_2 \eta_t + (1-\beta_2)||\nabla_\theta C(\theta)|_{\theta=\theta_t}||^2$$

$$\hat{v_{t+1}} = \frac{v_{t+1}}{1-\beta_1}$$

$$\hat{\eta_{t+1}} = \frac{\eta_{t+1}}{1-\beta_2}$$

These 1/(1-β) factors are to make v and η into unbiased estimators for the first and second moments of the gradient of C.

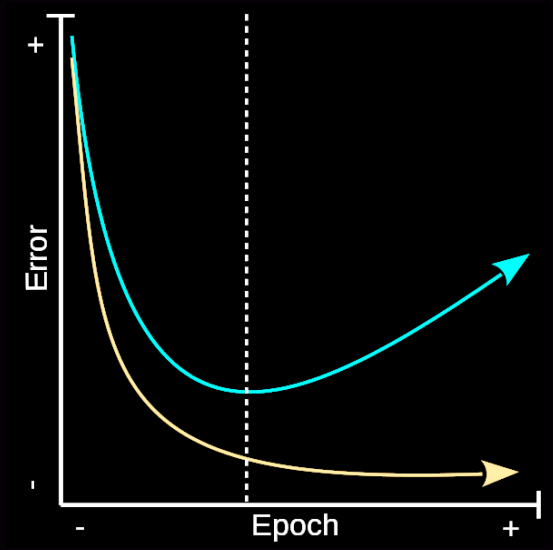
$$\Delta\theta_t = -\rho_0 \frac{\hat{v_{t+1}}}{\sqrt{\hat{\eta_{t+1}} + \epsilon}}$$

$$\theta_{t+1} = \theta_t + \Delta\theta_t$$

Nearly everyone will use defaults of β_1 = 0.9, β_2 = 0.99, ρ_0=0.01, ε=10e-7

Regularisation

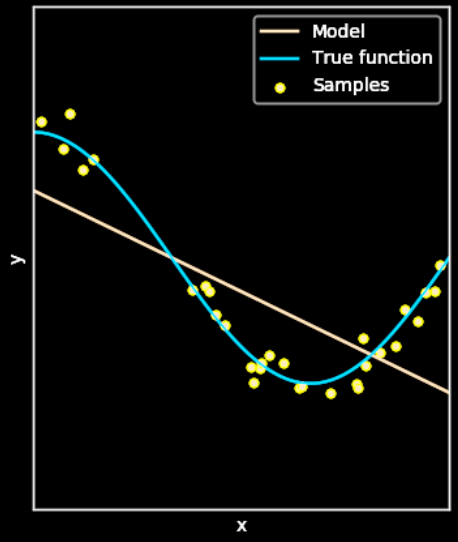
- Neural Networks have a propensity to overfit, meaning that improvements in the model's training performance can result in *degradations* in test performance.
- Networks fit to noise present in the training data — meaning they struggle to generalise to unseen examples.



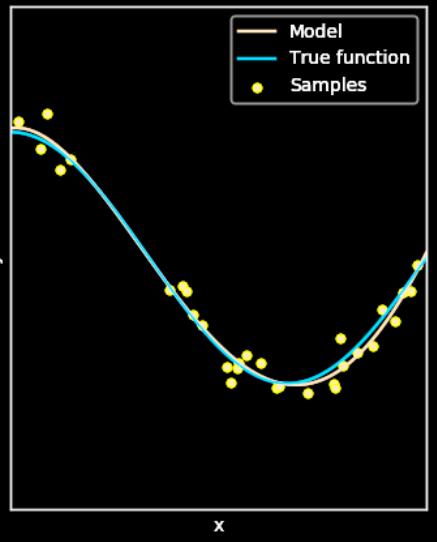
- validation performance
- training performance

Regularisation

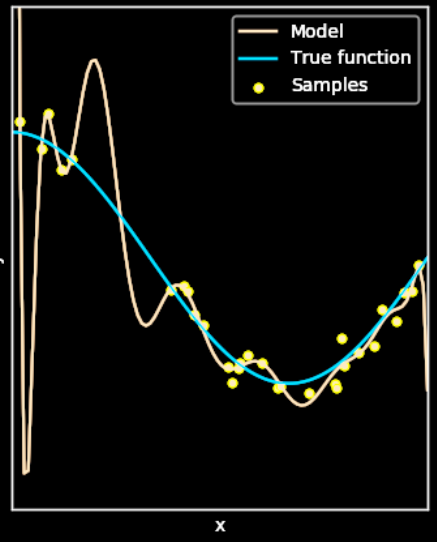
Underfitting



Optimum



Overfitting



Regularisation

Our goal is to prevent this from happening by penalising complex models.

L1 and L2 Regularisation

$$C_{L_1} = C + \lambda \sum_{k=0}^{M} |W^k|$$

- Broadly equivalent to Laplacian prior.

$$C_{L_2} = C + \lambda \sum_{k=0}^{M} ||W^k||_2$$

- Broadly equivalent to Normal prior.

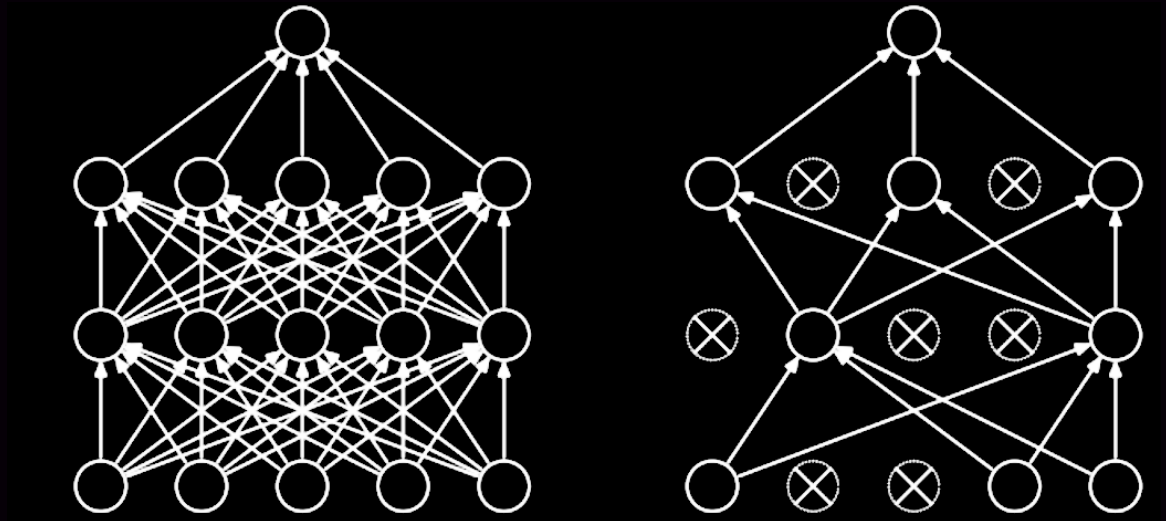
- Higher weight values have been associated with overfitting.

- Generally speaking if you want optimum prediction use L2. If you want network sparsity at some sacrifice of predictive discrimination use L1.

Dropout

Dropout - set some units' activations to zero during training with a given probability.

Seem super strange? Why do this?



Standard Network

After Dropout

- Dropout effectively superimposes multiple 'weaker/simpler' networks into a larger regularised network.
- Note that dropout doubles the number of epochs before convergence.
- Useful for very large networks.

Batch Normalisation

- Forces each layer of a neural network to learn a bit more independently to other layers.
  - Reduces *Internal Covariate Shift* (the change in the distribution of network activations due to the change parameters during training)

  - BN normalizes the output of a previous activation layer.
  - Simply normalizing each input of a layer may change what the layer can represent.
    - Normalizing the inputs of a sigmoid would **constrain them to the linear regime of the function.**
  - To address this, BN ensures that the transformation inserted in the network can represent the identity transform.

Batch Normalisation

- BN solves these issues by adding two trainable parameters to each layer, so the normalized output is multiplied by a “standard deviation” (gamma) and summed with a “mean” (beta).
- These parameters are learned along with the original model parameters, and restore the representation power of the network.

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

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

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

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

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

BN Algorithm

KERAS Example

```
# --- Load dataset from sklearn
digits = load_digits()

# --- Format features and labels
data = np.asarray(digits.data, dtype='float32')
target = np.asarray(digits.target, dtype='int32')

# --- Split into training and testing data
X_train, X_test, y_train, y_test = train_test_split(
    data, target, test_size=0.15, random_state=37)

# --- Normalize data
scaler = preprocessing.StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

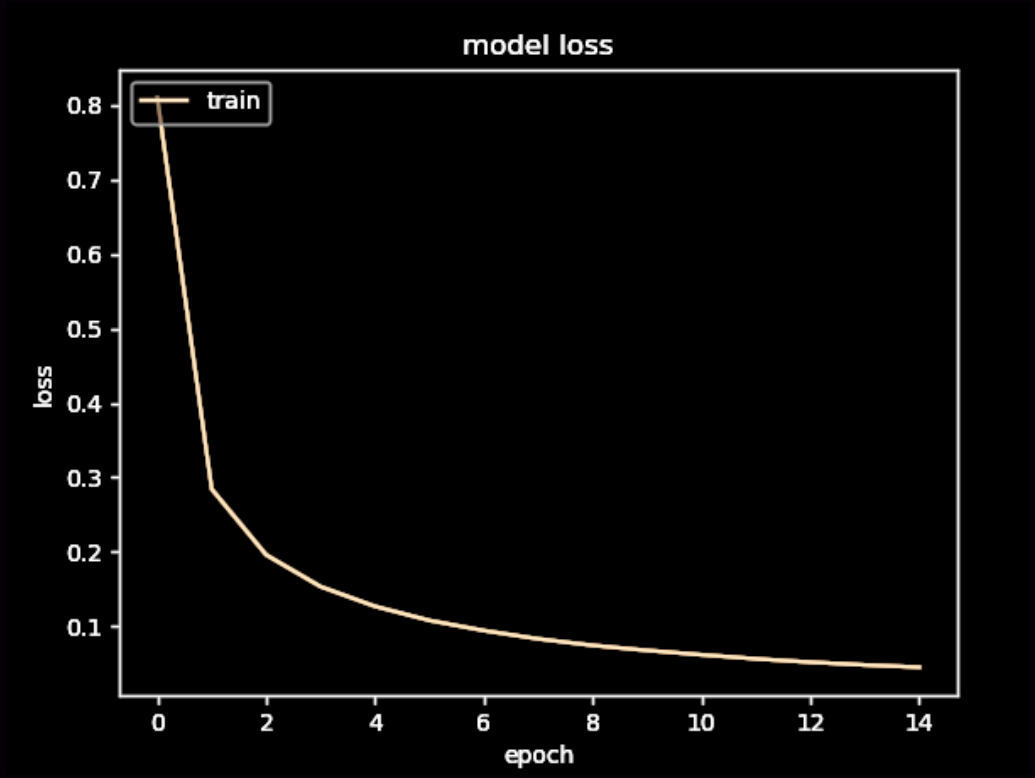
KERAS Example

```
Y_train = to_categorical(y_train)

N = X_train.shape[1]  # input size
H = 100  # hidden layer size or 'width'
K = 10  # output layer size, i.e number of classes
```

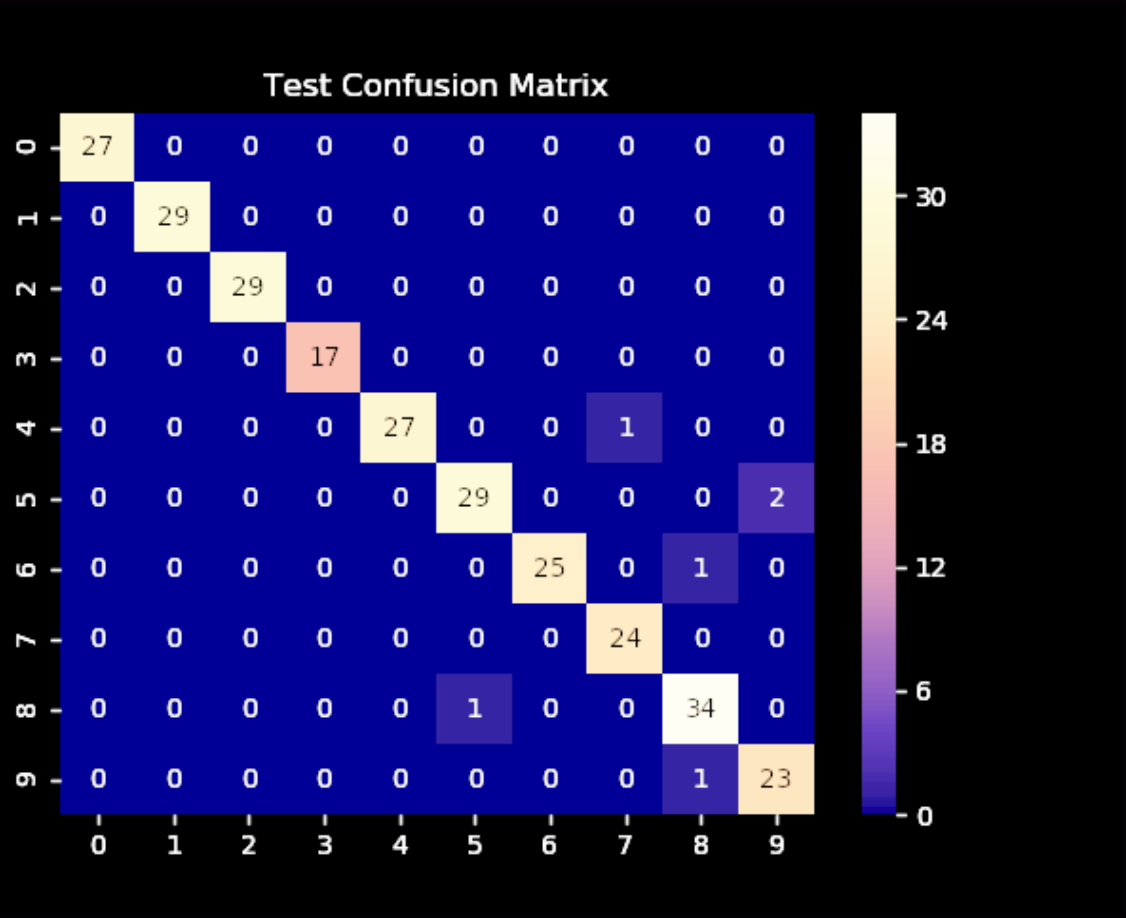
KERAS Example

```
# --- Keras sequential model
model = Sequential()
model.add(Dense(H, input_dim=N))
model.add(Activation("tanh"))
model.add(Dense(K))
model.add(Activation("softmax"))

# --- Compile and fit the model using SGD
model.compile(
    optimizer=optimizers.SGD(lr=0.1),
    loss='categorical_crossentropy',
    metrics=['accuracy'])

history = model.fit(X_train, Y_train, epochs=15, batch_size=32)
```

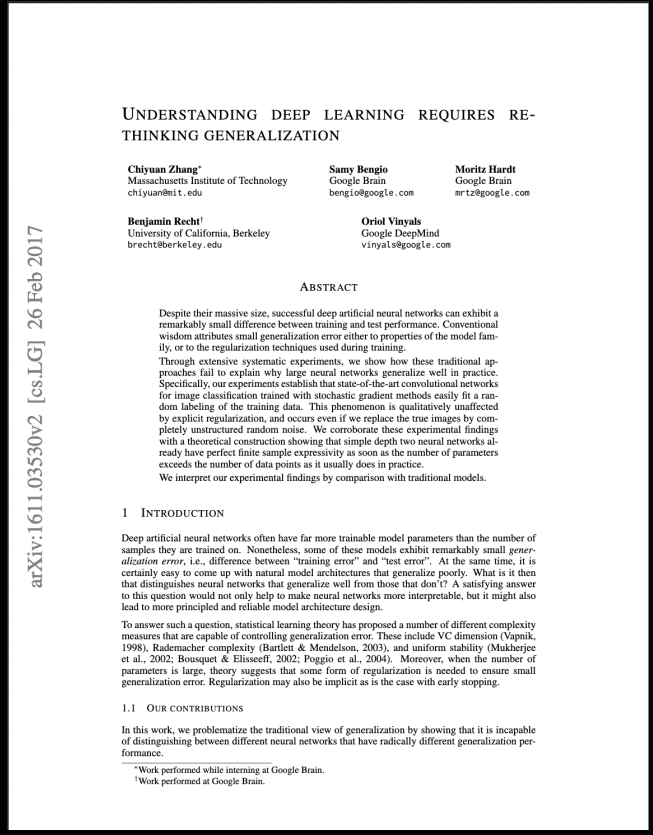
KERAS Example



KERAS Example



Paper: Understanding Deep Learning Requires Re-Thinking Generalization



UNDERSTANDING DEEP LEARNING REQUIRES RETHINKING GENERALIZATION

Chiyuan Zhang[*]
Massachusetts Institute of Technology
chiyuan@mit.edu

Samy Bengio
Google Brain
bengio@google.com

Moritz Hardt
Google Brain
mrtz@google.com

Benjamin Recht[†]
University of California, Berkeley
brecht@berkeley.edu

Oriol Vinyals
Google DeepMind
vinyals@google.com

ABSTRACT

Despite their massive size, successful deep artificial neural networks can exhibit a remarkably small difference between training and test performance. Conventional wisdom attributes small generalization error either to properties of the model family, or to the regularization techniques used during training.

Through extensive systematic experiments, we show how these traditional approaches fail to explain why large neural networks generalize well in practice. Specifically, our experiments establish that state-of-the-art convolutional networks for image classification trained with stochastic gradient methods easily fit a random labeling of the training data. This phenomenon is qualitatively unaffected by explicit regularization, and occurs even if we replace the true images by completely unstructured random noise. We corroborate these experimental findings with a theoretical construction showing that simple depth two neural networks already have perfect finite sample expressivity as soon as the number of parameters exceeds the number of data points as it usually does in practice.

We interpret our experimental findings by comparison with traditional models.

1 INTRODUCTION

Deep artificial neural networks often have far more trainable model parameters than the number of samples they are trained on. Nonetheless, some of these models exhibit remarkably small *generalization error*, i.e., difference between "training error" and "test error". At the same time, it is certainly easy to come up with natural model architectures that generalize poorly. What is it then that distinguishes neural networks that generalize well from those that don't? A satisfying answer to this question would not only help to make neural networks more interpretable, but it might also lead to more principled and reliable model architecture design.

To answer such a question, statistical learning theory has proposed a number of different complexity measures that are capable of controlling generalization error. These include VC dimension (Vapnik, 1998), Rademacher complexity (Bartlett & Mendelson, 2003), and uniform stability (Mukherjee et al., 2002; Bousquet & Elisseeff, 2002; Poggio et al., 2004). Moreover, when the number of parameters is large, theory suggests that some form of regularization is needed to ensure small generalization error. Regularization may also be implicit as is the case with early stopping.

1.1 OUR CONTRIBUTIONS

In this work, we problematize the traditional view of generalization by showing that it is incapable of distinguishing between different neural networks that have radically different generalization performance.

[*]Work performed while interning at Google Brain.
[†]Work performed at Google Brain.

Paper: Visualizing the Loss Landscape of Neural Nets

Visualizing the Loss Landscape of Neural Nets

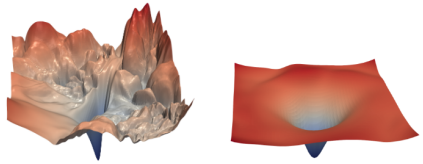
Hao Li[1], **Zheng Xu**[1], **Gavin Taylor**[2], **Christoph Studer**[3], **Tom Goldstein**[1]
[1]University of Maryland, College Park [2]United States Naval Academy [3]Cornell University
{haoli,xuzh,tomg}@cs.umd.edu, taylor@usna.edu, studer@cornell.edu

Abstract

Neural network training relies on our ability to find "good" minimizers of highly non-convex loss functions. It is well-known that certain network architecture designs (e.g., skip connections) produce loss functions that train easier, and well-chosen training parameters (batch size, learning rate, optimizer) produce minimizers that generalize better. However, the reasons for these differences, and their effect on the underlying loss landscape, are not well understood. In this paper, we explore the structure of neural loss functions, and the effect of loss landscapes on generalization, using a range of visualization methods. First, we introduce a simple "filter normalization" method that helps us visualize loss function curvature and make meaningful side-by-side comparisons between loss functions. Then, using a variety of visualizations, we explore how network architecture affects the loss landscape, and how training parameters affect the shape of minimizers.

1 Introduction

Training neural networks requires minimizing a high-dimensional non-convex loss function – a task that is hard in theory, but sometimes easy in practice. Despite the NP-hardness of training general neural loss functions [3], simple gradient methods often find global minimizers (parameter configurations with zero or near-zero training loss), even when data and labels are randomized before training [43]. However, this good behavior is not universal; the trainability of neural nets is highly dependent on network architecture design choices, the choice of optimizer, variable initialization, and a variety of other considerations. Unfortunately, the effect of each of these choices on the structure of the underlying loss surface is unclear. Because of the prohibitive cost of loss function evaluations (which requires looping over all the data points in the training set), studies in this field have remained predominantly theoretical.



(a) without skip connections

(b) with skip connections

Figure 1: The loss surfaces of ResNet-56 with/without skip connections. The proposed filter normalization scheme is used to enable comparisons of sharpness/flatness between the two figures.

32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

Paper: Adam: A Method for Stochastic Optimization

ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION

Diederik P. Kingma[*]
University of Amsterdam, OpenAI
dpkingma@openai.com

Jimmy Lei Ba[*]
University of Toronto
jimmy@psi.utoronto.ca

ABSTRACT

We introduce *Adam*, an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. The method is straightforward to implement, is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. The method is also appropriate for non-stationary objectives and problems with very noisy and/or sparse gradients. The hyper-parameters have intuitive interpretations and typically require little tuning. Some connections to related algorithms, on which *Adam* was inspired, are discussed. We also analyze the theoretical convergence properties of the algorithm and provide a regret bound on the convergence rate that is comparable to the best known results under the online convex optimization framework. Empirical results demonstrate that Adam works well in practice and compares favorably to other stochastic optimization methods. Finally, we discuss *AdaMax*, a variant of *Adam* based on the infinity norm.

1 INTRODUCTION

Stochastic gradient-based optimization is of core practical importance in many fields of science and engineering. Many problems in these fields can be cast as the optimization of some scalar parameterized objective function requiring maximization or minimization with respect to its parameters. If the function is differentiable w.r.t. its parameters, gradient descent is a relatively efficient optimization method, since the computation of first-order partial derivatives w.r.t. all the parameters is of the same computational complexity as just evaluating the function. Often, objective functions are stochastic. For example, many objective functions are composed of a sum of subfunctions evaluated at different subsamples of data; in this case optimization can be made more efficient by taking gradient steps w.r.t. individual subfunctions, i.e. stochastic gradient descent (SGD) or ascent. SGD proved itself as an efficient and effective optimization method that was central in many machine learning success stories, such as recent advances in deep learning (Deng et al., 2013; Krizhevsky et al., 2012; Hinton & Salakhutdinov, 2006; Hinton et al., 2012a; Graves et al., 2013). Objectives may also have other sources of noise than data subsampling, such as dropout (Hinton et al., 2012b) regularization. For all such noisy objectives, efficient stochastic optimization techniques are required. The focus of this paper is on the optimization of stochastic objectives with high-dimensional parameters spaces. In these cases, higher-order optimization methods are ill-suited, and discussion in this paper will be restricted to first-order methods.

We propose *Adam*, a method for efficient stochastic optimization that only requires first-order gradients with little memory requirement. The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients; the name *Adam* is derived from adaptive moment estimation. Our method is designed to combine the advantages of two recently popular methods: AdaGrad (Duchi et al., 2011), which works well with sparse gradients, and RMSProp (Tieleman & Hinton, 2012), which works well in on-line and non-stationary settings; important connections to these and other stochastic optimization methods are clarified in section 5. Some of Adam's advantages are that the magnitudes of parameter updates are invariant to rescaling of the gradient, its stepsizes are approximately bounded by the stepsize hyperparameter, it does not require a stationary objective, it works with sparse gradients, and it naturally performs a form of step size annealing.

[*]Equal contribution. Author ordering determined by coin flip over a Google Hangout.

Acknowledgments

- Grant Sanderson for figures
- Gabriel Goh for the momentum simulations
- Chris Holmes and Gil McVean for arranging this seminar series