Hello, My name is Becca and I am a PhD student at UC Berkeley. This paper is joint work with Ashia Wilson, Mitchell Stern, and Benjamin Recht also at UC Berkeley, as well as Nati Srebro, front guard of the NIPS rigor police.
Deep learning is an extremely powerful technology, but it has a drawback, which is that in order to actually use it, one has to contend with a sea of decisions. Given a task and dataset, we have to decide what architecture and what optimization algorithm to use. But both of these choices come with a host of sub-choices.

Ultimately, we end up with an intractable design space that is a tuning nightmare. Obviously, we can’t explore the entire space, [there might be latent dependencies, it is wayyyyy too complicated], so what we end up doing is leaving the majority of the variables set to fixed defaults, while changing only a small subset of them.

A popular default in deep learning is to choose the optimization algorithm to be adaptive, which includes Adam, RMSProp, and AdaGrad. Adam in particular is extremely popular and has been cited over 5100 times in the past three years.
Deep learning is an extremely powerful technology, but it has a drawback, which is that in order to actually use it, one has to contend with a sea of decisions. Given a task and dataset, we have to decide what architecture and what optimization algorithm to use. But both of these choices come with a host of sub-choices.

Ultimately, we end up with an intractable design space that is a tuning nightmare. Obviously, we can’t explore the entire space, [there might be latent dependencies, it is wayyyy too complicated], so what we end up doing is leaving the majority of the variables set to fixed defaults, while changing only a small subset of them.

A popular default in deep learning is to choose the optimization algorithm to be adaptive, which includes Adam, RMSProp, and AdaGrad. Adam in particular is extremely popular and has been cited over 5100 times in the past three years.
Deep learning is an extremely powerful technology, but it has a drawback, which is that in order to actually use it, one has to contend with a sea of decisions. Given a task and dataset, we have to decide what architecture and what optimization algorithm to use. But both of these choices come with a host of sub-choices.

Ultimately, we end up with an intractable design space that is a tuning nightmare.

Obviously, we can’t explore the entire space, [there might be latent dependencies, it is wayyy too complicated], so what we end up doing is leaving the majority of the variables set to fixed defaults, while changing only a small subset of them.

A popular default in deep learning is to choose the optimization algorithm to be adaptive, which includes Adam, RMSProp, and AdaGrad. Adam in particular is extremely popular and has been cited over 5100 times in the past three years.
Deep learning is an extremely powerful technology, but it has a drawback, which is that in order to actually use it, one has to contend with a sea of decisions. Given a task and dataset, we have to decide what architecture and what optimization algorithm to use. But both of these choices come with a host of sub-choices.

Ultimately, we end up with an intractable design space that is a tuning nightmare. Obviously, we can’t explore the entire space, [there might be latent dependencies, it is wayyyy too complicated], so what we end up doing is leaving the majority of the variables set to fixed defaults, while changing only a small subset of them.

A popular default in deep learning is to choose the optimization algorithm to be adaptive, which includes Adam, RMSProp, and AdaGrad. Adam in particular is extremely popular and has been cited over 5100 times in the past three years.
Deep learning is an extremely powerful technology, but it has a drawback, which is that in order to actually use it, one has to contend with a sea of decisions. Given a task and dataset, we have to decide what architecture and what optimization algorithm to use. But both of these choices come with a host of sub-choices.

Ultimately, we end up with an intractable design space that is a tuning nightmare. Obviously, we can’t explore the entire space, [there might be latent dependencies, it is wayyy too complicated], so what we end up doing is leaving the majority of the variables set to fixed defaults, while changing only a small subset of them.

A popular default in deep learning is to choose the optimization algorithm to be adaptive, which includes Adam, RMSProp, and AdaGrad. Adam in particular is extremely popular and has been cited over 5100 times in the past three years.
Deep learning is an extremely powerful technology, but it has a drawback, which is that in order to actually use it, one has to contend with a sea of decisions. Given a task and dataset, we have to decide what architecture and what optimization algorithm to use. But both of these choices come with a host of sub-choices. Ultimately, we end up with an intractable design space that is a tuning nightmare. Obviously, we can’t explore the entire space, [there might be latent dependencies, it is wayyy too complicated], so what we end up doing is leaving the majority of the variables set to fixed defaults, while changing only a small subset of them.

A popular default in deep learning is to choose the optimization algorithm to be adaptive, which includes Adam, RMSProp, and AdaGrad. Adam in particular is extremely popular and has been cited over 5100 times in the past three years.
This talk will focus on why adaptive algorithms are not always a good choice.
We will begin by reviewing some of the algorithms used in deep learning; We start with SGD, which works by computing the gradient on a batch of data and stepping along the negative direction of the gradient. There are two main ingredients that are typically added to SGD.

The first is momentum, which has many types, but a common type, which is the heavy ball method, adds a scaled difference between the current and the previous iterate.

The second is adaptivity. These are methods that are adaptive to the scale of the parameters, or in other words ignore the relative scale of the parameters.

One way to achieve adaptivity is to multiply the gradient with a matrix $H_k^{-1}$ where $H_k$ is a diagonal matrix computed using a weighted sum of the previous gradients.

$H_k$ can be interpreted as a per-parameter learning rate, which might be useful if you want a larger learning rate for sparse, rare features. But it also might have the effect of ignoring important scale information contained in the parameters.

Adam is a combination of adaptivity and momentum. The formula here can be derived from the original Adam update by unrolling the recursion.
We will begin by reviewing some of the algorithms used in deep learning; We start with SGD, which works by computing the gradient on a batch of data and stepping along the negative direction of the gradient. There are two main ingredients that are typically added to SGD.

The first is momentum, which has many types, but a common type, which is the heavy ball method, adds a scaled difference between the current and the previous iterate.

The second is adaptivity. These are methods that are adaptive to the scale of the parameters, or in other words ignore the relative scale of the parameters.

One way to achieve adaptivity is to multiply the gradient with a matrix $H_k^{-1}$ where $H_k$ is a diagonal matrix computed using a weighted sum of the previous gradients.

$H_k$ can be interpreted as a per-parameter learning rate, which might be useful if you want a larger learning rate for sparse, rare features. But it also might have the effect of ignoring important scale information contained in the parameters.

Adam is a combination of adaptivity and momentum. The formula here can be derived from the original Adam update by unrolling the recursion.
We will begin by reviewing some of the algorithms used in deep learning; We start with SGD, which works by computing the gradient on a batch of data and stepping along the negative direction of the gradient. There are two main ingredients that are typically added to SGD.

The first is momentum, which has many types, but a common type, which is the heavy ball method, adds a scaled difference between the current and the previous iterate.

The second is adaptivity. These are methods that are adaptive to the scale of the parameters, or in other words ignore the relative scale of the parameters. One way to achieve adaptivity is to multiply the gradient with a matrix $H_k^{-1}$ where $H_k$ is a diagonal matrix computed using a weighted sum of the previous gradients.

$H_k$ can be interpreted as a per-parameter learning rate, which might be useful if you want a larger learning rate for sparse, rare features. But it also might have the effect of ignoring important scale information contained in the parameters.

Adam is a combination of adaptivity and momentum. The formula here can be derived from the original Adam update by unrolling the recursion.
We will begin by reviewing some of the algorithms used in deep learning; We start with SGD, which works by computing the gradient on a batch of data and stepping along the negative direction of the gradient. There are two main ingredients that are typically added to SGD.

The first is momentum, which has many types, but a common type, which is the heavy ball method, adds a scaled difference between the current and the previous iterate.

The second is adaptivity. These are methods that are adaptive to the scale of the parameters, or in other words ignore the relative scale of the parameters.

One way to achieve adaptivity is to multiply the gradient with a matrix $H_k^{-1}$ where $H_k$ is a diagonal matrix computed using a weighted sum of the previous gradients.

$H_k$ can be interpreted as a per-parameter learning rate, which might be useful if you want a larger learning rate for sparse, rare features. But it also might have the effect of ignoring important scale information contained in the parameters.

Adam is a combination of adaptivity and momentum. The formula here can be derived from the original Adam update by unrolling the recursion.
We will begin by reviewing some of the algorithms used in deep learning; We start with SGD, which works by computing the gradient on a batch of data and stepping along the negative direction of the gradient. There are two main ingredients that are typically added to SGD.

The first is momentum, which has many types, but a common type, which is the heavy ball method, adds a scaled difference between the current and the previous iterate.

The second is adaptivity. These are methods that are adaptive to the scale of the parameters, or in other words ignore the relative scale of the parameters.

One way to achieve adaptivity is to multiply the gradient with a matrix $H^{-1}_k$ where $H_k$ is a diagonal matrix computed using a weighted sum of the previous gradients.

$H_k = \text{diag} \left( \sqrt{\sum_{i=1}^{k} \eta_i \nabla f(x_i)^2} \right)$

Adaptive to scale of parameters
• Can be interpreted as a per-parameter learning rate

Adam is a combination of adaptivity and momentum. The formula here can be derived from the original Adam update by unrolling the recursion.
We will begin by reviewing some of the algorithms used in deep learning; We start with SGD, which works by computing the gradient on a batch of data and stepping along the negative direction of the gradient. There are two main ingredients that are typically added to SGD.

The first is momentum, which has many types, but a common type, which is the heavy ball method, adds a scaled difference between the current and the previous iterate.

The second is adaptivity. These are methods that are adaptive to the scale of the parameters, or in other words ignore the relative scale of the parameters. One way to achieve adaptivity is to multiply the gradient with a matrix $H_k^{-1}$ where $H_k$ is a diagonal matrix computed using a weighted sum of the previous gradients.

$H_k$ can be interpreted as a per-parameter learning rate, which might be useful if you want a larger learning rate for sparse, rare features. But it also might have the effect of ignoring important scale information contained in the parameters.

Adam is a combination of adaptivity and momentum. The formula here can be derived from the original Adam update by unrolling the recursion.
We will begin by reviewing some of the algorithms used in deep learning; We start with SGD, which works by computing the gradient on a batch of data and stepping along the negative direction of the gradient. There are two main ingredients that are typically added to SGD.

The first is momentum, which has many types, but a common type, which is the heavy ball method, adds a scaled difference between the current and the previous iterate.

The second is adaptivity. These are methods that are adaptive to the scale of the parameters, or in other words ignore the relative scale of the parameters.

One way to achieve adaptivity is to multiply the gradient with a matrix $H_k^{-1}$ where $H_k$ is a diagonal matrix computed using a weighted sum of the previous gradients.

$H_k$ can be interpreted as a per-paramater learning rate, which might be useful if you want a larger learning rate for sparse, rare features. But it also might have the effect of ignoring important scale information contained in the parameters.

Adam is a combination of adaptivity and momentum. The formula here can be derived from the original Adam update by unrolling the recursion.
A central question we ask is whether “adaptivity is a good idea for deep learning?”

To answer this question, we perform several experiments across vision and natural language processing tasks.

Next, we analyze an illustrative binary classification problem which helps us explain the circumstances under which adaptive methods can fail.
A central question we ask is whether “adaptivity is a good idea for deep learning?”

To answer this question, we perform several experiments across vision and natural language processing tasks.

Next, we analyze an illustrative binary classification problem which helps us explain the circumstances under which adaptive methods can fail.
A central question we ask is whether “adaptivity is a good idea for deep learning?”

To answer this question, we perform several experiments across vision and natural language processing tasks.

Next, we analyze an illustrative binary classification problem which helps us explain the circumstances under which adaptive methods can fail.
A central question we ask is whether “adaptivity is a good idea for deep learning?”

To answer this question, we perform several experiments across vision and natural language processing tasks.

Next, we analyze an illustrative binary classification problem which helps us explain the circumstances under which adaptive methods can fail.
Generative Parsing on Penn Treebank

Training Set

What follows can be seen as a summary of our main experimental observation. The task is Generative Parsing on Penn Treebank dataset.
On the left, we plot a training curve for Adam and SGD, that we obtained after extensive hyper parameter tuning.
Adam in red is outperforming SGD in blue.
On the right, is the performance of the algorithms on the validation set.
Critically, Adam significantly underperforms on unseen data. Performance on this dataset is our metric for learning and what we care about most.
Also, the difference between Adam and SGD in terms of perplexity was equivalent to several years of improvement in parsing research.
We were able to replicate the observation that SGD performed the best on the test set across three other architectures and datasets. We also found that this behavior was not only specific to Adam and SGD, but also to adaptive and non-adaptive methods.
What follows can be seen as a summary of our main experimental observation. The task is Generative Parsing on Penn Treebank dataset. On the left, we plot a training curve for Adam and SGD, that we obtained after extensive hyper parameter tuning. Adam in red is outperforming SGD in blue.

On the right, is the performance of the algorithms on the validation set. Critically, Adam significantly underperforms on unseen data. Performance on this dataset is our metric for learning and what we care about most. Also, the difference between Adam and SGD in terms of perplexity was equivalent to several years of improvement in parsing research.

We were able to replicate the observation that SGD performed the best on the test set across three other architectures and datasets. We also found that this behavior was not only specific to Adam and SGD, but also to adaptive and non-adaptive methods.
In more detail, I'll describe how we ran these experiments.
We repeated each experiment five times from random initialization,
and made minimal changes to the online codebases containing each architecture.
I'd like to thank each of the authors for publicly releasing their models.
We ran 4 tasks, which consisted of one convolutional network from vision and three lstms from nlp.
For each one of these tasks we ran 5 algorithms, 2 non-adaptive, SGD and Heavy Ball, 3 adaptive methods, Adam, Adagrad and RMSProp.
We also included Adam with the default learning rate and no learning rate decay, to see how well it would do.
Of the non adaptive methods, SGD and Heavy ball tended to track each other's behavior, but after tuning, SGD was always slightly better than SGD with momentum.
Of the adaptive methods, Adam tended to be best, though RMSProp was a close runner up.
For the purposes of conciseness, in the remainder of the talk, we will use Adam as the representative adaptive algorithm and SGD as the representative non-adaptive algorithm, though generally, what we say holds across adaptive and non-adaptive methods.
The complete set of results are in our paper.
In more detail, I’ll describe how we ran these experiments.
We repeated each experiment five times from random initialization, and made minimal changes to the online codebases containing each architecture.
I’d like to thank each of the authors for publicly releasing their models.
We ran 4 tasks, which consisted of one convolutional network from vision and three lstms from nlp.
For each one of these tasks we ran 5 algorithms, 2 non-adaptive, SGD and Heavy Ball, 3 adaptive methods, Adam, Adagrad and RMSProp.
We also included Adam with the default learning rate and no learning rate decay, to see how well it would do.
Of the non adaptive methods, SGD and Heavy ball tended to track each other’s behavior, but after tuning, SGD was always slightly better than SGD with momentum.
Of the adaptive methods, Adam tended to be best, though RMSProp was a close runner up.
For the purposes of conciseness, in the remainder of the talk, we will use Adam as the representative adaptive algorithm and SGD as the representative non-adaptive algorithm, though generally, what we say holds across adaptive and non-adaptive methods.
The complete set of results are in our paper.
Deep Learning Experiments

- Repeat 5x with random initialization
- Minimal changes to online codebases

In more detail, I’ll describe how we ran these experiments.

We repeated each experiment five times from random initialization, and made minimal changes to the online codebases containing each architecture.

I’d like to thank each of the authors for publicly releasing their models.

We ran 4 tasks, which consisted of one convolutional network from vision and three lstms from nlp.

For each one of these tasks we ran 5 algorithms, 2 non-adaptive, SGD and Heavy Ball, 3 adaptive methods, Adam, Adagrad and RMSProp.

We also included Adam with the default learning rate and no learning rate decay, to see how well it would do.

Of the non adaptive methods, SGD and Heavy ball tended to track each other's behavior, but after tuning, SGD was always slightly better than SGD with momentum.

Of the adaptive methods, Adam tended to be best, though RMSProp was a close runner up.

For the purposes of conciseness, in the remainder of the talk, we will use Adam as the representative adaptive algorithm and SGD as the representative non-adaptive algorithm, though generally, what we say holds across adaptive and non-adaptive methods.

The complete set of results are in our paper.
Deep Learning Experiments

- Repeat 5x with random initialization
- Minimal changes to online codebases

1. Sergey Zagoruyko:
   https://github.com/szagoruyko/cifar.torch

2. Justin Johnson & Andrej Karparthy:
   https://github.com/jcjohnson/torch-rnn

3. James Cross & Liang Huang:
   https://github.com/jhcross/span-parser

4. Do Kook Choe & Eugene Charniak:

In more detail, I'll describe how we ran these experiments.
We repeated each experiment five times from random initialization,
and made minimal changes to the online codebases containing each architecture.
I'd like to thank each of the authors for publicly releasing their models.
We ran 4 tasks, which consisted of one convolutional network from vision and three lstms from nlp.
For each one of these tasks we ran 5 algorithms, 2 non-adaptive, SGD and Heavy Ball, 3 adaptive methods, Adam, Adagrad and RMSProp.
We also included Adam with the default learning rate and no learning rate decay, to see how well it would do.
Of the non adaptive methods, SGD and Heavy ball tended to track each other's behavior, but after tuning, SGD was always slightly better than SGD with momentum.
Of the adaptive methods, Adam tended to be best, though RMSProp was a close runner up.
For the purposes of conciseness, in the remainder of the talk, we will use Adam as the representative adaptive algorithm and SGD as the representative non-adaptive algorithm, though generally, what we say holds across adaptive and non-adaptive methods.
The complete set of results are in our paper.
Deep Learning Experiments

- Repeat 5x with random initialization
- Minimal changes to online codebases

In more detail, I'll describe how we ran these experiments.
We repeated each experiment five times from random initialization, and made minimal changes to the online codebases containing each architecture.
I'd like to thank each of the authors for publicly releasing their models.
We ran 4 tasks, which consisted of one convolutional network from vision and three lstms from nlp.
For each one of these tasks we ran 5 algorithms, 2 non-adaptive, SGD and Heavy Ball, 3 adaptive methods, Adam, Adagrad and RMSProp.
We also included Adam with the default learning rate and no learning rate decay, to see how well it would do.
Of the non adaptive methods, SGD and Heavy ball tended to track each other's behavior, but after tuning, SGD was always slightly better than SGD with momentum.
Of the adaptive methods, Adam tended to be best, though RMSProp was a close runner up.
For the purposes of conciseness, in the remainder of the talk, we will use Adam as the representative adaptive algorithm and SGD as the representative non-adaptive algorithm, though generally, what we say holds across adaptive and non-adaptive methods.
The complete set of results are in our paper.
In more detail, I'll describe how we ran these experiments. We repeated each experiment five times from random initialization, and made minimal changes to the online codebases containing each architecture.

I'd like to thank each of the authors for publicly releasing their models.

We ran 4 tasks, which consisted of one convolutional network from vision and three lstms from nlp. For each one of these tasks we ran 5 algorithms, 2 non-adaptive, SGD and Heavy Ball, 3 adaptive methods, Adam, Adagrad and RMSProp. We also included Adam with the default learning rate and no learning rate decay, to see how well it would do.

Of the non adaptive methods, SGD and Heavy ball tended to track each other's behavior, but after tuning, SGD was always slightly better than SGD with momentum.

Of the adaptive methods, Adam tended to be best, though RMSProp was a close runner up. For the purposes of conciseness, in the remainder of the talk, we will use Adam as the representative adaptive algorithm and SGD as the representative non-adaptive algorithm, though generally, what we say holds across adaptive and non-adaptive methods.

The complete set of results are in our paper.
In more detail, I’ll describe how we ran these experiments.

We repeated each experiment five times from random initialization, and made minimal changes to the online codebases containing each architecture.

I’d like to thank each of the authors for publicly releasing their models.

We ran 4 tasks, which consisted of one convolutional network from vision and three lstms from nlp.

For each one of these tasks we ran 5 algorithms, 2 non-adaptive, SGD and Heavy Ball, 3 adaptive methods, Adam, Adagrad and RMSProp.

We also included Adam with the default learning rate and no learning rate decay, to see how well it would do.

Of the non adaptive methods, SGD and Heavy ball tended to track each other’s behavior, but after tuning, SGD was always slightly better than SGD with momentum.

Of the adaptive methods, Adam tended to be best, though RMSProp was a close runner up.

For the purposes of conciseness, in the remainder of the talk, we will use Adam as the representative adaptive algorithm and SGD as the representative non-adaptive algorithm, though generally, what we say holds across adaptive and non-adaptive methods.

The complete set of results are in our paper.
In more detail, I'll describe how we ran these experiments.

We repeated each experiment five times from random initialization, and made minimal changes to the online codebases containing each architecture.

I'd like to thank each of the authors for publicly releasing their models.

We ran 4 tasks, which consisted of one convolutional network from vision and three lstms from nlp.

For each one of these tasks we ran 5 algorithms, 2 non-adaptive, SGD and Heavy Ball, 3 adaptive methods, Adam, Adagrad and RMSProp.

We also included Adam with the default learning rate and no learning rate decay, to see how well it would do.

Of the non adaptive methods, SGD and Heavy ball tended to track each other's behavior, but after tuning, SGD was always slightly better than SGD with momentum.

Of the adaptive methods, Adam tended to be best, though RMSProp was a close runner up.

For the purposes of conciseness, in the remainder of the talk, we will use Adam as the representative adaptive algorithm and SGD as the representative non-adaptive algorithm, though generally, what we say holds across adaptive and non-adaptive methods.

The complete set of results are in our paper.
In more detail, I’ll describe how we ran these experiments.

We repeated each experiment five times from random initialization, and made minimal changes to the online codebases containing each architecture.

I’d like to thank each of the authors for publicly releasing their models.

We ran 4 tasks, which consisted of one convolutional network from vision and three lstms from nlp.

For each one of these tasks we ran 5 algorithms, 2 non-adaptive, SGD and Heavy Ball, 3 adaptive methods, Adam, Adagrad and RMSProp.

We also included Adam with the default learning rate and no learning rate decay, to see how well it would do.

Of the non adaptive methods, SGD and Heavy ball tended to track each other’s behavior, but after tuning, SGD was always slightly better than SGD with momentum.

Of the adaptive methods, Adam tended to be best, though RMSProp was a close runner up.

For the purposes of conciseness, in the remainder of the talk, we will use Adam as the representative adaptive algorithm and SGD as the representative non-adaptive algorithm, though generally, what we say holds across adaptive and non-adaptive methods.

The complete set of results are in our paper.
In more detail, I’ll describe how we ran these experiments. We repeated each experiment five times from random initialization, and made minimal changes to the online codebases containing each architecture.

I’d like to thank each of the authors for publicly releasing their models.

We ran 4 tasks, which consisted of one convolutional network from vision and three lstms from nlp.

For each one of these tasks we ran 5 algorithms, 2 non-adaptive, SGD and Heavy Ball, 3 adaptive methods, Adam, Adagrad and RMSProp.

We also included Adam with the default learning rate and no learning rate decay, to see how well it would do.

Of the non adaptive methods, SGD and Heavy ball tended to track each other’s behavior, but after tuning, SGD was always slightly better than SGD with momentum.

Of the adaptive methods, Adam tended to be best, though RMSProp was a close runner up.

For the purposes of conciseness, in the remainder of the talk, we will use Adam as the representative adaptive algorithm and SGD as the representative non-adaptive algorithm, though generally, what we say holds across adaptive and non-adaptive methods.

The complete set of results are in our paper.
On Cifar 10, we ran a deep convolutional network with batch normalization and dropout.
In the first 50 epochs, Adam in red is outperforming SGD in blue on the training set.
But this initial progress is misleading.
The story changes when we extend the time scale out to 250 epochs. Ultimately, SGD surpasses Adam and achieves the lowest training error.
The kinks you see in this graph are caused by decaying the step size, which I'll talk about more in a bit.
On the test set, which is again our metric for learning and what we really care about, SGD outperforms Adam.
And, Adam with its default parameters is doing terribly.
On Cifar 10, we ran a deep convolutional network with batch normalization and dropout. In the first 50 epochs, Adam in red is outperforming SGD in blue on the training set. But this initial progress is misleading.

The story changes when we extend the time scale out to 250 epochs. Ultimately, SGD surpasses Adam and achieves the lowest training error. The kinks you see in this graph are caused by decaying the step size, which I’ll talk about more in a bit. On the test set, which is again our metric for learning and what we really care about, SGD outperforms Adam. And, Adam with its default parameters is doing terribly.
On Cifar 10, we ran a deep convolutional network with batch normalization and dropout.
In the first 50 epochs, Adam in red is outperforming SGD in blue on the training set.
But this initial progress is misleading.

The story changes when we extend the time scale out to 250 epochs. Ultimately, SGD surpasses Adam and achieves the lowest training error.
The kinks you see in this graph are caused by decaying the step size, which I'll talk about more in a bit.

On the test set, which is again our metric for learning and what we really care about, SGD outperforms Adam.
And, Adam with its default parameters is doing terribly.
On Cifar 10, we ran a deep convolutional network with batch normalization and dropout.
In the first 50 epochs, Adam in red is outperforming SGD in blue on the training set.
But this initial progress is misleading.
The story changes when we extend the time scale out to 250 epochs. Ultimately, SGD surpasses Adam and achieves the lowest training error.
The kinks you see in this graph are caused by decaying the step size, which I'll talk about more in a bit.
On the test set, which is again our metric for learning and what we really care about, SGD outperforms Adam.
And, Adam with its default parameters is doing terribly.
On Cifar 10, we ran a deep convolutional network with batch normalization and dropout.
In the first 50 epochs, Adam in red is outperforming SGD in blue on the training set.
But this initial progress is misleading.
The story changes when we extend the time scale out to 250 epochs. Ultimately, SGD surpasses Adam and achieves the lowest training error.
The kinks you see in this graph are caused by decaying the step size, which I'll talk about more in a bit.
On the test set, which is again our metric for learning and what we really care about, SGD outperforms Adam.
And, Adam with its default parameters is doing terribly.
**Experiment:** Cifar-10 Image Classification with VGG+BN+Dropout Model

On Cifar 10, we ran a deep convolutional network with batch normalization and dropout. In the first 50 epochs, Adam in red is outperforming SGD in blue on the training set. But this initial progress is misleading. The story changes when we extend the time scale out to 250 epochs. Ultimately, SGD surpasses Adam and achieves the lowest training error. The kinks you see in this graph are caused by decaying the step size, which I'll talk about more in a bit. On the test set, which is again our metric for learning and what we really care about, SGD outperforms Adam. And, Adam with its default parameters is doing terribly.
On Cifar 10, we ran a deep convolutional network with batch normalization and dropout. In the first 50 epochs, Adam in red is outperforming SGD in blue on the training set. But this initial progress is misleading.

The story changes when we extend the time scale out to 250 epochs. Ultimately, SGD surpasses Adam and achieves the lowest training error. The kinks you see in this graph are caused by decaying the step size, which I'll talk about more in a bit.

On the test set, which is again our metric for learning and what we really care about, SGD outperforms Adam. And, Adam with its default parameters is doing terribly.
In every experiment, Adam is faster initially, but SGD has best test error.

I don’t have time to show the other two tasks, but
The important takeaway from our experiments is that in every experiment, Adam is faster initially on the training set, but SGD has better test error.

So, what is going on here, if SGD is this great, why is Adam so popular? One issue might be how the step size is chosen for SGD
How we chose the step size

In our experiments, we evaluated a logarithmically-spaced grid of at least five step sizes. If the best performance was ever at one of the extremes of the grid, we would try new grid points so that the best performance was contained in the middle of the parameters we tried.

For example, if we initially tried this range of step sizes and found that 2 was the best performing, we would have tried 4 to see if performance was improved, and so on.

We also found that decaying the step size is critical for all algorithms.

We tried two decay schemes,

One was a validation decay scheme where we would keep track of the best validation error and decay if we saw that the validation error was going up.

The other was a fixed decay scheme where we would decay at a fixed interval, such as every 25 epochs.

In both schemes, we tuned the amount that you needed to decay.

We found that while using a decay scheme was necessary, the two decay schemes we tried had equivalent performance.
In our experiments, we evaluated a logarithmically-spaced grid of at least five step sizes.
If the best performance was ever at one of the extremes of the grid, we would try new grid points so that the best performance was contained in the middle of the parameters we tried.
For example, if we initially tried this range of step sizes and found that 2 was the best performing, we would have tried 4 to see if performance was improved, and so on.
We also found that decaying the step size is critical for all algorithms.

We tried two decay schemes,

One was a validation decay scheme where we would keep track of the best validation error and decay if we saw that the validation error was going up.
The other was a fixed decay scheme where we would decay at a fixed interval, such as every 25 epochs.

In both schemes, we tuned the amount that you needed to decay.

We found that while using a decay scheme was necessary, the two decay schemes we tried had equivalent performance.
How we chose the step size

Logarithmically spaced grid of at least five step sizes

In our experiments, we evaluated a logarithmically-spaced grid of at least five step sizes. If the best performance was ever at one of the extremes of the grid, we would try new grid points so that the best performance was contained in the middle of the parameters we tried. For example, if we initially tried this range of step sizes and found that 2 was the best performing, we would have tried 4 to see if performance was improved, and so on.

We also found that decaying the step size is critical for all algorithms. We tried two decay schemes:

One was a validation decay scheme where we would keep track of the best validation error and decay if we saw that the validation error was going up. The other was a fixed decay scheme where we would decay at a fixed interval, such as every 25 epochs.

In both schemes, we tuned the amount that you needed to decay.

We found that while using a decay scheme was necessary, the two decay schemes we tried had equivalent performance.
How we chose the step size

Logarithmically spaced grid of at least five step sizes

In our experiments, we evaluated a logarithmically-spaced grid of at least five step sizes.

If the best performance was ever at one of the extremes of the grid, we would try new grid points so that the best performance was contained in the middle of the parameters we tried.

For example, if we initially tried this range of step sizes and found that 2 was the best performing, we would have tried 4 to see if performance was improved, and so on.

We also found that decaying the step size is critical for all algorithms.

We tried two decay schemes,

One was a validation decay scheme where we would keep track of the best validation error and decay if we saw that the validation error was going up.

The other was a fixed decay scheme where we would decay at a fixed interval, such as every 25 epochs.

In both schemes, we tuned the amount that you needed to decay.

We found that while using a decay scheme was necessary, the two decay schemes we tried had equivalent performance.
In our experiments, we evaluated a logarithmically-spaced grid of at least five step sizes.
If the best performance was ever at one of the extremes of the grid, we would try new grid points so that the best performance was contained in the middle of the parameters we tried.

For example, if we initially tried this range of step sizes and found that 2 was the best performing, we would have tried 4 to see if performance was improved, and so on.

We also found that decaying the step size is critical for all algorithms.

We tried two decay schemes,

One was a validation decay scheme where we would keep track of the best validation error and decay if we saw that the validation error was going up.

The other was a fixed decay scheme where we would decay at a fixed interval, such as every 25 epochs.

In both schemes, we tuned the amount that you needed to decay.

We found that while using a decay scheme was necessary, the two decay schemes we tried had equivalent performance.
In our experiments, we evaluated a logarithmically-spaced grid of at least five step sizes. If the best performance was ever at one of the extremes of the grid, we would try new grid points so that the best performance was contained in the middle of the parameters we tried.

For example, if we initially tried this range of step sizes and found that 2 was the best performing, we would have tried 4 to see if performance was improved, and so on.

We also found that decaying the step size is critical for all algorithms.

We tried two decay schemes,

One was a validation decay scheme where we would keep track of the best validation error and decay if we saw that the validation error was going up.

The other was a fixed decay scheme where we would decay at a fixed interval, such as every 25 epochs.

In both schemes, we tuned the amount that you needed to decay.

We found that while using a decay scheme was necessary, the two decay schemes we tried had equivalent performance.
In our experiments, we evaluated a logarithmically-spaced grid of at least five step sizes. If the best performance was ever at one of the extremes of the grid, we would try new grid points so that the best performance was contained in the middle of the parameters we tried. For example, if we initially tried this range of step sizes and found that 2 was the best performing, we would have tried 4 to see if performance was improved, and so on.

We also found that decaying the step size is critical for all algorithms. We tried two decay schemes,

- Validation Decay: decay if validation set is not best seen
- Fixed Decay: decay at fixed intervals, e.g. every 25 epochs

In both schemes, we tuned the amount that you needed to decay. We found that while using a decay scheme was necessary, the two decay schemes we tried had equivalent performance.
One of the conclusions we had from these experiments was that Adam needs the same amount of tuning as SGD. This goes against the conventional wisdom that Adam works well out of the box. To show this more clearly, we plotted the best test error achieved by SGD and Adam in 250 epochs on Cifar-10 versus a scaled step size. You can interpret the scale on the x axis as how much you would have to multiply the best step size by to get this test error.

This graph illustrates that there is an optimal step-size for each algorithm, and tuning is needed for both algorithms to find it. If the difference between the two algorithms isn’t the amount of variance around the optimal step size, it could be how close the default step size is to the best step size.

To see this, we can highlight the default step sizes for SGD. These default step sizes come from the pytorch and tensor flow tutorials. The step size recommended for Adam is only a small factor off from the best step size, whereas the SGD default step size is often orders of magnitude off. What we really need is a better default step size for SGD. The behavior of SGD on the plot can help us find one. SGD gets good test error right up until the point where it starts to diverge, which is when it gets 90% error on cifar-10. It is really easy to detect when SGD starts to diverge.

This is an age old heuristic, but a good experimental rule of thumb is to pick the largest step size for SGD that does not diverge. This worked well across all of our experiments.
Both **Adam** and **SGD** need tuning

Cifar10 VGG best test error found in 250 epochs

![Graph showing test error versus step size for SGD and Adam](chart.png)

One of the conclusions we had from these experiments was that Adam needs the same amount of tuning as SGD. This goes against the conventional wisdom that Adam works well out of the box.

To show this more clearly, we plotted the best test error achieved by SGD and Adam in 250 epochs on Cifar-10 versus a scaled step size. You can interpret the scale on the x axis as how much you would have to multiply the best step size by to get this test error.

This graph illustrates that there is an optimal step-size for each algorithm, and tuning is needed for both algorithms to find it.

If the difference between the two algorithms isn’t the amount of variance around the optimal step size, it could be how close the default step size is to the best step size.

To see this, we can highlight the default step sizes for SGD. These default step sizes come from the pytorch and tensor flow tutorials.

The step size recommended for Adam is only a small factor off from the best step size, whereas the SGD default step size is often orders of magnitude off.

What we really need is a better default step size for SGD. The behavior of SGD on the plot can help us find one. SGD gets good test error right up until the point where it starts to diverge, which is when it gets 90% error on cifar-10. It is really easy to detect when SGD starts to diverge.

This is an age old heuristic, but a good experimental rule of thumb is to pick the largest step size for SGD that does not diverge. This worked well across all of our experiments.
Both Adam and SGD need tuning

Cifar10 VGG best test error found in 250 epochs

- SGD default step size is often orders of magnitude off

One of the conclusions we had from these experiments was that Adam needs the same amount of tuning as SGD. This goes against the conventional wisdom that Adam works well out of the box.

To show this more clearly, we plotted the best test error achieved by SGD and Adam in 250 epochs on Cifar-10 versus a scaled step size. You can interpret the scale on the x axis as how much you would have to multiply the best step size by to get this test error.

This graph illustrates that there is an optimal step-size for each algorithm, and tuning is needed for both algorithms to find it. If the difference between the two algorithms isn’t the amount of variance around the optimal step size, it could be how close the default step size is to the best step size.

To see this, we can highlight the default step sizes for SGD. These default step sizes come from the pytorch and tensor flow tutorials. The step size recommended for Adam is only a small factor off from the best step size, whereas the SGD default step size is often orders of magnitude off.

What we really need is a better default step size for SGD. The behavior of SGD on the plot can help us find one. SGD gets good test error right up until the point where it starts to diverge, which is when it gets 90% error on cifar-10. It is really easy to detect when SGD starts to diverge.

This is an age old heuristic, but a good experimental rule of thumb is to pick the largest step size for SGD that does not diverge. This worked well across all of our experiments.
One of the conclusions we had from these experiments was that Adam needs the same amount of tuning as SGD. This goes against the conventional wisdom that Adam works well out of the box.

To show this more clearly, we plotted the best test error achieved by SGD and Adam in 250 epochs on Cifar-10 versus a scaled step size. You can interpret the scale on the x axis as how much you would have to multiply the best step size by to get this test error.

This graph illustrates that there is an optimal step-size for each algorithm, and tuning is needed for both algorithms to find it. If the difference between the two algorithms isn’t the amount of variance around the optimal step size, it could be how close the default step size is to the best step size.

To see this, we can highlight the default step sizes for SGD. These default step sizes come from the pytorch and tensorflow tutorials. The step size recommended for Adam is only a small factor off from the best step size, whereas the SGD default step size is often orders of magnitude off.

What we really need is a better default step size for SGD. The behavior of SGD on the plot can help us find one. SGD gets good test error right up until the point where it starts to diverge, which is when it gets 90% error on cifar-10. It is really easy to detect when SGD starts to diverge.

This is an age old heuristic, but a good experimental rule of thumb is to pick the largest step size for SGD that does not diverge. This worked well across all of our experiments.
Both Adam and SGD need tuning

Cifar10 VGG best test error found in 250 epochs

- SGD default step size is often orders of magnitude off
- Pick the largest step size for SGD that does not diverge

One of the conclusions we had from these experiments was that Adam needs the same amount of tuning as SGD. This goes against the conventional wisdom that Adam works well out of the box.

To show this more clearly, we plotted the best test error achieved by SGD and Adam in 250 epochs on Cifar-10 versus a scaled step size. You can interpret the scale on the x axis as how much you would have to multiply the best step size by to get this test error.

This graph illustrates that there is an optimal step-size for each algorithm, and tuning is needed for both algorithms to find it.

If the difference between the two algorithms isn’t the amount of variance around the optimal step size, it could be how close the default step size is to the best step size.

To see this, we can highlight the default step sizes for SGD. These default step sizes come from the pytorch and tensorflow tutorials.

The step size recommended for Adam is only a small factor off from the best step size, whereas the SGD default step size is often orders of magnitude off.

What we really need is a better default step size for SGD. The behavior of SGD on the plot can help us find one. SGD gets good test error right up until the point where it starts to diverge, which is when it gets 90% error on cifar-10. It is really easy to detect when SGD starts to diverge.

This is an age old heuristic, but a good experimental rule of thumb is to pick the largest step size for SGD that does not diverge. This worked well across all of our experiments.
To try to understand these experimental observations, we analyzed the behavior of Adam and SGD on a binary least squares classification problem.

We're doing this because if the algorithms don't work well in the linear setting, we have no reason to expect them to work generally.

My advisor, Ben, calls this the Recht Linearization Principle
To try to understand these experimental observations, we analyzed the behavior of Adam and SGD on a binary least squares classification problem.

We're doing this because if the algorithms don't work well in the linear setting, we have no reason to expect them to work generally.

My advisor, Ben, calls this the Recht Linearization Principle.
To try to understand these experimental observations, we analyzed the behavior of Adam and SGD on a binary least squares classification problem.

We're doing this because if the algorithms don't work well in the linear setting, we have no reason to expect them to work generally.

My advisor, Ben, calls this the Recht Linearization Principle
Specifically, we consider the underdetermined least square setting.

We want to allow for overly complicated functions, or the potential for overfitting.

How do you do this in the linear case? You make the number of features $d$ much larger than the number of examples $n$.

This, also means that there are many possible solutions to the problem, possibly an infinite number if the problem can be solved exactly.

So, which solution does each algorithm find when initialized?

SGD and its variants will find the minimum norm solution. This is also the solution with the maximum margin separating data from the two classes.

The solution found by Adam is more complicated. In particular, which solution adaptive methods will find will depend on which step size is used, which is not true for SGD.

One thing we can say is that if there exists a solution with uniform parameters, which means the magnitude of each parameter is equal, this is the solution that Adam finds.

The specific uniform weight solution that Adam will find is proportional to the sign $(X^T y)$. 
Specifically, we consider the underdetermined least square setting.
We want to allow for overly complicated functions, or the potential for overfitting.
How do you do this in the linear case? You make the number of features $d$ much larger than the number of examples $n$.
This, also means that there are many possible solutions to the problem, possibly an infinite number if the problem can be solved exactly.
So, which solution does each algorithm find when initialized?
SGD and its variants will find the minimum norm solution. This is also the solution with the maximum margin separating data from the two classes.
The solution found by Adam is more complicated. In particular, which solution adaptive methods will find will depend on which step size is used, which is not true for SGD.
One thing we can say is that if there exists a solution with
uniform parameters, which means the magnitude of each parameter is equal, this is the solution that Adam finds
The specific uniform weight solution that Adam will find is proportional to the sign $(X^T y)$. 
Specifically, we consider the underdetermined least square setting. We want to allow for overly complicated functions, or the potential for overfitting. How do you do this in the linear case? You make the number of features $d$ much larger than the number of examples $n$. This also means that there are many possible solutions to the problem, possibly an infinite number if the problem can be solved exactly.

So, which solution does each algorithm find when initialized? SGD and its variants will find the minimum norm solution. This is also the solution with the maximum margin separating data from the two classes. The solution found by Adam is more complicated. In particular, which solution adaptive methods will find will depend on which step size is used, which is not true for SGD.

One thing we can say is that if there exists a solution with uniform parameters, which means the magnitude of each parameter is equal, this is the solution that Adam finds. The specific uniform weight solution that Adam will find is proportional to the sign ($X^Ty$).
Specifically, we consider the underdetermined least square setting. We want to allow for overly complicated functions, or the potential for overfitting. How do you do this in the linear case? You make the number of features $d$ much larger than the number of examples $n$. This, also means that there are many possible solutions to the problem, possibly an infinite number if the problem can be solved exactly. So, which solution does each algorithm find when initialized? SGD and its variants will find the minimum norm solution. This is also the solution with the maximum margin separating data from the two classes. The solution found by Adam is more complicated. In particular, which solution adaptive methods will find will depend on which step size is used, which is not true for SGD. One thing we can say is that if there exists a solution with uniform parameters, which means the magnitude of each parameter is equal, this is the solution that Adam finds. The specific uniform weight solution that Adam will find is proportional to the sign $(X^T y)$.
Specifically, we consider the underdetermined least square setting. We want to allow for overly complicated functions, or the potential for overfitting. How do you do this in the linear case? You make the number of features $d$ much larger than the number of examples $n$. This, also means that there are many possible solutions to the problem, possibly an infinite number if the problem can be solved exactly. So, which solution does each algorithm find when initialized? SGD and its variants will find the minimum norm solution. This is also the solution with the maximum margin separating data from the two classes. The solution found by Adam is more complicated. In particular, which solution adaptive methods will find will depend on which step size is used, which is not true for SGD.

One thing we can say is that if there exists a solution with uniform parameters, which means the magnitude of each parameter is equal, this is the solution that Adam finds. The specific uniform weight solution that Adam will find is proportional to the sign $(X^T y)$. 
The closed form solutions for Adam and SGD helps us to understand the kinds of examples on which Adam might fail to find a solution that generalizes well. We construct the example by first picking the label. Once we know the label, we can build the feature matrix deterministically depending on the sign of the label. For i = 1...n we will sample the label $y_i$ to be 1 with probability $\frac{1}{2} + \epsilon$, which means on average there are slightly more positive examples than negative examples.

We then construct $x_i$ so that the first feature is the class label. The next 2 features are always equal to 1. After this, there are set of extremely sparse features where each example $i$ gets to populate its own block of features in a location unique to $x_i$. What the block contains as well as the size of the block will depend on the label $y_i$. If the class label is 1, then there is 1 such feature. If the class label is -1, there are 5.

The unique location is important because it means that the features are not necessarily predictive of the label for new data. We can show that on this example, Adagrad, RMSProp, and Adam always classify new data as belonging to the positive class, and SGD finds a good separator with zero error.

So, what is it about this example that causes Adam to fail? Here, we have a correct predictor, which is sparse and a lot of uninformative rare features. In particular, Adam seems to be putting undue weight on the sparse rare features which are not predictive outside of the training set. Therefore, it ends up always classifying new data as positive.

Though not shown on this slide, we have an algebraic proof of this which is written in our full paper.
The closed form solutions for Adam and SGD helps us to understand the kinds of examples on which Adam might fail to find a solution that generalizes well. We construct the example by first picking the label. Once we know the label, we can build the feature matrix deterministically depending on the sign of the label. For $i = 1 \ldots n$ we will sample the label $y_i$ to be 1 with probability $\frac{1}{2} + \epsilon$, which means on average there are slightly more positive examples than negative examples.

We then construct $x_i$ so that the first feature is the class label. The next 2 features are always equal to 1. After this, there are set of extremely sparse features where each example $i$ gets to populate its own block of features in a location unique to $x_i$. What the block contains as well as the size of the block will depend on the label $y_i$. If the class label is 1, then there is 1 such feature. If the class label is -1, there are 5.

The unique location is important because it means that the features are not necessarily predictive of the label for new data. We can show that on this example, Adagrad, RMSProp, and Adam always classify new data as belonging to the positive class, and SGD finds a good separator with zero error.

So, what is it about this example that causes Adam to fail? Here, we have a correct predictor, which is sparse and a lot of uninformative rare features. In particular, Adam seems to be putting undue weight on the sparse rare features which are not predictive outside of the training set. Therefore, it ends up always classifying new data as positive.

Though not shown on this slide, we have an algebraic proof of this which is written in our full paper.
Example where Adaptive Methods Fail to Generalize

Sample label $y_i \in \{-1, +1\}$, equal to +1 with probability $\frac{1}{2} + \epsilon$

$x_i = \begin{bmatrix} y_i & 1 & 1 & 0 & 0 & \ldots & 1 & 1 & 1 & 1 & \ldots & 0 & 0 \end{bmatrix}$

The closed form solutions for Adam and SGD help us to understand the kinds of examples on which Adam might fail to find a solution that generalizes well.

We construct the example by first picking the label. Once we know the label, we can build the feature matrix deterministically depending on the sign of the label.

For $i = 1 \ldots n$ we will sample the label $y_i$ to be 1 with probability $\frac{1}{2} + \epsilon$, which means on average there are slightly more positive examples than negative examples.

We then construct $x_i$ so that the first feature is the class label. The next 2 features are always equal to 1. After this, there are set of extremely sparse features where each example $i$ gets to populate its own block of features in a location unique to $x_i$. What the block contains as well as the size of the block will depend on the label $y_i$. If the class label is 1, then there is 1 such feature. If the class label is -1, there are 5.

The unique location is important because it means that the features are not necessarily predictive of the label for new data.

We can show that on this example, Adagrad, RMSProp, and Adam always classify new data as belonging to the positive class, and SGD finds a good separator with zero error.

So, what is it about this example that causes Adam to fail? Here, we have a correct predictor, which is sparse and a lot of uninformative rare features. In particular, Adam seems to be putting undue weight on the sparse rare features which are not predictive outside of the training set. Therefore, it ends up always classifying new data as positive.

Though not shown on this slide, we have an algebraic proof of this which is written in our full paper.
Example where Adaptive Methods Fail to Generalize

Sample label $y_i \in \{-1, +1\}$, equal to $+1$ with probability $\frac{1}{2} + \epsilon$

$$x_i = \begin{bmatrix} y_i \ 1 \ 1 \ 0 \ 0 \ 0 \ \ldots \ \ 1 \ 1 \ 1 \ 1 \ 1 \ \ldots \ 0 \ 0 \end{bmatrix}$$

- Five 1s for $y_i = -1$
- One 1 for $y_i = +1$
- Different location for each example

The closed form solutions for Adam and SGD helps us to understand the kinds of examples on which Adam might fail to find a solution that generalizes well. We construct the example by first picking the label. Once we know the label, we can build the feature matrix deterministically depending on the sign of the label. For $i = 1 \ldots n$ we will sample the label $y_i$ to be 1 with probability $\frac{1}{2} + \epsilon$, which means on average there are slightly more positive examples than negative examples.

We then construct $x_i$ so that the first feature is the class label. The next 2 features are always equal to 1. After this, there are set of extremely sparse features where each example $i$ gets to populate its own block of features in a location unique to $x_i$. What the block contains as well as the size of the block will depend on the label $y_i$. If the class label is 1, then there is 1 such feature. If the class label is -1, there are 5.

The unique location is important because it means that the features are not necessarily predictive of the label for new data.

We can show that on this example, Adagrad, RMSProp, and Adam always classify new data as belonging to the positive class, and SGD finds a good separator with zero error.

So, what is it about this example that causes Adam to fail? Here, we have a correct predictor, which is sparse and a lot of uninformative rare features. In particular, Adam seems to be putting undue weight on the sparse rare features which are not predictive outside of the training set. Therefore, it ends up always classifying new data as positive.

Though not shown on this slide, we have an algebraic proof of this which is written in our full paper.
Example where Adaptive Methods Fail to Generalize

Sample label $y_i \in \{-1, +1\}$, equal to $+1$ with probability $\frac{1}{2} + \epsilon$

\[
\begin{array}{cccccccc}
  y_i & 1 & 1 & 0 & 0 & \ldots & 1 & 1 & 1 & 1 & \ldots & 0 & 0 \\
\end{array}
\]

- Five 1s for $y_i = -1$
- One 1 for $y_i = +1$
- Different location for each example

\begin{itemize}
  \item Adagrad, RMSProp, and Adam always classify new data as positive
  \item SGD finds a good separator with zero error
\end{itemize}

The closed form solutions for Adam and SGD help us to understand the kinds of examples on which Adam might fail to find a solution that generalizes well.

We construct the example by first picking the label. Once we know the label, we can build the feature matrix deterministically depending on the sign of the label. For $i = 1...n$ we will sample the label $y_i$ to be 1 with probability $\frac{1}{2} + \epsilon$, which means on average there are slightly more positive examples than negative examples.

We then construct $x_i$ so that the first feature is the class label. The next 2 features are always equal to 1. After this, there are set of extremely sparse features where each example $i$ gets to populate its own block of features in a location unique to $x_i$. What the block contains as well as the size of the block will depend on the label $y_i$. If the class label is 1, then there is 1 such feature. If the class label is -1, there are 5.

The unique location is important because it means that the features are not necessarily predictive of the label for new data.

We can show that on this example, Adagrad, RMSProp, and Adam always classify new data as belonging to the positive class, and SGD finds a good separator with zero error.

So, what is it about this example that causes Adam to fail? Here, we have a correct predictor, which is sparse and a lot of uninformative rare features. In particular, Adam seems to be putting undue weight on the sparse rare features which are not predictive outside of the training set. Therefore, it ends up always classifying new data as positive.

Though not shown on this slide, we have an algebraic proof of this which is written in our full paper.
Example where Adaptive Methods Fail to Generalize

Sample label $y_i \in \{-1, +1\}$ equal to $+1$ with probability $\frac{1}{2} + \epsilon$

$\begin{pmatrix}
  y_i & 1 & 1 & 0 & 0 & \ldots & 1 & 1 & 1 & 1 & \ldots & 0 & 0
\end{pmatrix}$

- Adagrad, RMSProp, and Adam always classify new data as positive
- SGD finds a good separator with zero error

Adam fails because it puts undue weight on sparse, rare features, which are uninformative.

Five 1s for $y_i = -1$
One 1 for $y_i = +1$
Different location for each example

The closed form solutions for Adam and SGD helps us to understand the kinds of examples on which Adam might fail to find a solution that generalizes well.

We construct the example by first picking the label. Once we know the label, we can build the feature matrix deterministically depending on the sign of the label. For $i = 1 \ldots n$ we will sample the label $y_i$ to be 1 with probability $\frac{1}{2} + \epsilon$, which means on average there are slightly more positive examples than negative examples.

We then construct $x_i$ so that the first feature is the class label. The next 2 features are always equal to 1. After this, there are set of extremely sparse features where each example $i$ gets to populate its own block of features in a location unique to $x_i$. What the block contains as well as the size of the block will depend on the label $y_i$. If the class label is 1, then there is 1 such feature. If the class label is -1, there are 5.

The unique location is important because it means that the features are not necessarily predictive of the label for new data.

We can show that on this example, Adagrad, RMSProp, and Adam always classify new data as belonging to the positive class, and SGD finds a good separator with zero error.

So, what is it about this example that causes Adam to fail? Here, we have a correct predictor, which is sparse and a lot of uninformative rare features. In particular, Adam seems to be putting undue weight on the sparse rare features which are not predictive outside of the training set. Therefore, it ends up always classifying new data as positive.

Though not shown on this slide, we have an algebraic proof of this which is written in our full paper.
In conclusion, in our deep learning experiments we found that:

- SGD (+momentum) outperform adaptive methods on unseen data
- Adaptive methods display fast initial progress, then plateau
- Both Adam and SGD need tuning
- Adam overfits to sparse uninformative features; SGD avoids such overfitting
In conclusion, in our deep learning experiments we found that

- SGD (+momentum) outperform adaptive methods on unseen data
- Adaptive methods display fast initial progress, then plateau
- Both Adam and SGD need tuning
- Adam overfits to sparse uninformative features; SGD avoids such overfitting
In conclusion, in our deep learning experiments we found that

- SGD (+momentum) outperforms adaptive methods on unseen data
- Adaptive methods display fast initial progress, then plateau
- Both Adam and SGD need tuning
- Adam overfits to sparse uninformative features; SGD avoids such overfitting
In conclusion, in our deep learning experiments we found that:

- SGD (+momentum) outperform adaptive methods on unseen data
- Adaptive methods display fast initial progress, then plateau
- Both Adam and SGD need tuning
- Adam overfits to sparse uninformative features; SGD avoids such overfitting
In conclusion, in our deep learning experiments we found that

- SGD (+momentum) outperforms adaptive methods on unseen data
- Adaptive methods display fast initial progress, then plateau
- Both Adam and SGD need tuning
- Adam overfits to sparse uninformative features; SGD avoids such overfitting
Where we do go from here? Our results suggest we should revisit using Adam as a default. We also would like to encourage the community to try to replicate our experimental results.

Follow-up work we are particularly interested in is model selection. We noticed that Adam is popular in choosing models for deep learning. We plan to explore whether models we obtain using SGD to choose hyper-parameters of the architecture are different than architectures chosen using Adam, and how this effects the architecture’s test performance.
Where we do go from here? Our results suggest we should revisit using Adam as a default. We also would like to encourage the community to try to replicate our experimental results.

Follow-up work we are particularly interested in is model selection. We noticed that Adam is popular in choosing models for deep learning. We plan to explore whether models we obtain using SGD to choose hyper-parameters of the architecture are different than architectures chosen using Adam, and how this effects the architecture’s test performance.
Where do we go from here? Our results suggest we should revisit using Adam as a default. We also would like to encourage the community to try to replicate our experimental results.

Follow-up work we are particularly interested in is model selection. We noticed that Adam is popular in choosing models for deep learning. We plan to explore whether models we obtain using SGD to choose hyper-parameters of the architecture are different than architectures chosen using Adam, and how this affects the architecture's test performance.
Where do we go from here? Our results suggest we should revisit using Adam as a default. We also would like to encourage the community to try to replicate our experimental results.

Follow-up work we are particularly interested in is model selection. We noticed that Adam is popular in choosing models for deep learning. We plan to explore whether models we obtain using SGD to choose hyper-parameters of the architecture are different than architectures chosen using Adam, and how this effects the architecture’s test performance.
Where do we go from here?

1. Revisit using Adam as a default
2. Replicate our results across other tasks and datasets

Where do go from here? Our results suggest we should revisit using Adam as a default. We also would like to encourage the community to try to replicate our experimental results.

Follow-up work we are particularly interested in is model selection. We noticed that Adam is popular in choosing models for deep learning. We plan to explore whether models we obtain using SGD to choose hyper-parameters of the architecture are different than architectures chosen using Adam, and how this effects the architecture’s test performance.
Where do we go from here?

1. Revisit using Adam as a default
2. Replicate our results across other tasks and datasets

What about model selection?

Where do we go from here? Our results suggest we should revisit using Adam as a default. We also would like to encourage the community to try to replicate our experimental results.

Follow-up work we are particularly interested in is model selection. We noticed that Adam is popular in choosing models for deep learning. We plan to explore whether models we obtain using SGD to choose hyper-parameters of the architecture are different than architectures chosen using Adam, and how this effects the architecture’s test performance.
Where do we go from here?

1. Revisit using Adam as a default
2. Replicate our results across other tasks and datasets

What about model selection?

Adam is popular for choosing models in DL. Is this wise?

Where do go from here? Our results suggest we should revisit using Adam as a default. We also would like to encourage the community to try to replicate our experimental results.

Follow-up work we are particularly interested in is model selection. We noticed that Adam is popular in choosing models for deep learning. We plan to explore whether models we obtain using SGD to choose hyper-parameters of the architecture are different than architectures chosen using Adam, and how this effects the architecture’s test performance.
Where do we go from here?

1. Revisit using Adam as a default
2. Replicate our results across other tasks and datasets

What about model selection?

Adam is popular for choosing models in DL. Is this wise?

Where do go from here? Our results suggest we should revisit using Adam as a default. We also would like to encourage the community to try to replicate our experimental results.

Follow-up work we are particularly interested in is model selection. We noticed that Adam is popular in choosing models for deep learning. We plan to explore whether models we obtain using SGD to choose hyper-parameters of the architecture are different than architectures chosen using Adam, and how this effects the architecture’s test performance.
In conclusion, deep learning involves many choices that we have to make. In this talk I showed you that the optimization algorithm is an important choice, and that adaptive methods are not always a good choice for deep learning models.
In conclusion, deep learning involves many choices that we have to make. In this talk I showed you that the optimization algorithm is an important choice, and that adaptive methods are not always a good choice for deep learning models.
In conclusion, deep learning involves many choices that we have to make. In this talk I showed you that the optimization algorithm is an important choice, and that adaptive methods are not always a good choice for deep learning models.